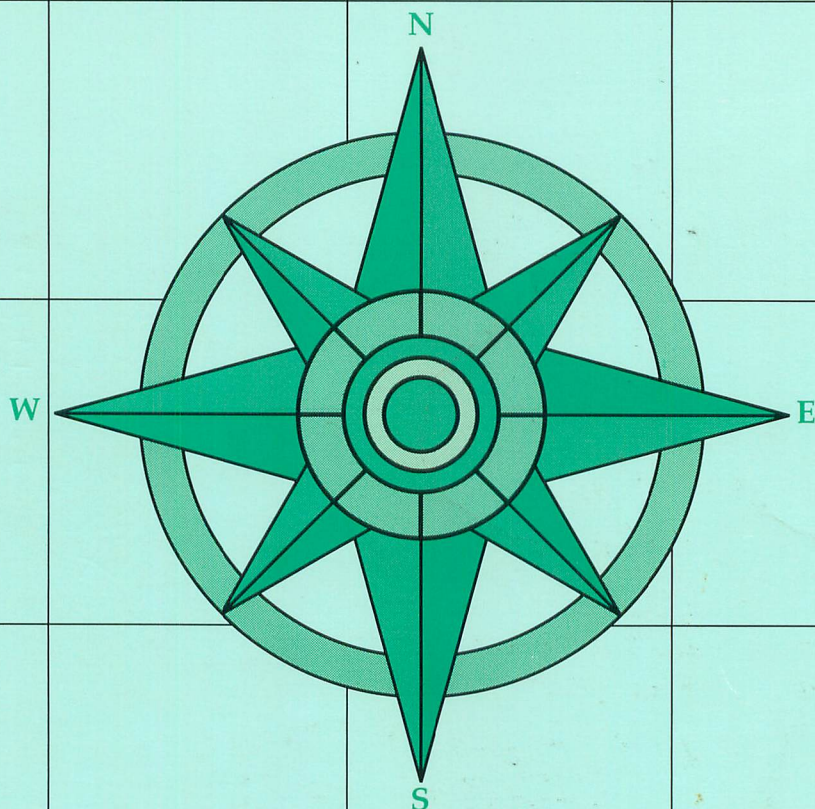


WShell2.0

Professional Command Shell
for Amiga[®] Computers



Wishful Thinking
Development Corp.



The Beachcomber's Guide to the WShell

Version 2.0

Copyright © 1988, 1991 William S. Hawes
All Rights Reserved

Copyright Notice

WShell software and documentation are copyright ©1988,1991 by William S. Hawes. All rights reserved. No part of the software or documentation may be reproduced, transmitted, translated into other languages, posted to a network, or distributed in any way without the express written permission of the author.

Disclaimer

This product is offered for sale "as is" with no representation of fitness for any particular purpose. The user assumes all risks and responsibilities related to its use. The material within is believed to be accurate, but the author reserves the right to make changes to the software or documentation without notice.

Distribution

WShell software and documentation may be purchased from:

Wishful Thinking Development Corp.
P.O. Box 308
Maynard, MA 01754
(508) 568-8695

Please direct orders or inquiries about this product to the above address. Other products available include ARexx, an implementation of the REXX language.

About ...

WShell is a command shell designed as a much-enhanced but highly compatible alternative to the Amiga's built-in Shell. It was written in assembly language using the CAPE assembler. The manual was written in American English using the TurboText editor, and was set in T_EX using AmigaT_EX. This is a 100% Amiga product.

Trademarks

ARexx and WShell are trademarks of Wishful Thinking Development Corp.; Amiga is a registered trademark of Commodore-Amiga, Inc.; AmigaDOS, Intuition, and Amiga Workbench are trademarks of Commodore-Amiga, Inc.

Table of Contents

The Beachcomber's Guide to the WShell

Chapter 1. Introducing WShell 2.0	1
1 Compatibility	1
2 Using this Manual	2
1 Typographic Conventions	3
3 Installing the WShell Software	3
1 Library Files	3
2 Utility Programs	4
3 Handlers	4
4 Configuration Files	4
5 Command Files	4
6 The Startup Sequence	5
7 Startup Scripts	5
8 Environment Variables	5
9 Activating the DisplayHandler	6
10 Activating Filename-Completion	6
11 Activating the PathHandler	6
4 Testing the WShell	6
5 Using WShell from Icons	7
Chapter 2. The Command Environment and Shell Conventions	9
1 The Command Environment	9
1 The Current Directory	9
2 Input and Output Streams	9
3 Stack Size	10
4 PROGDIR: Program Directory	10
2 Command Structure	10
1 Program Names	10
2 Redirection Specifiers	10
3 Command Arguments	11
4 Piping Specifiers	11
5 Background Specifiers	11
3 Comments, Quoting, and Escape Characters	12
1 Quote Character	12
2 Escape Character	12
3 Comments	12
4 Continuations	12
4 Variable Expansion and Backticked Commands	13
1 Variable Expansion	13
2 Backticked Commands	13
5 Environment and Local Variables	14

6 The Search Path	14
1 The Previous Command	15
2 Resident and Built-In Commands	15
3 REXX-Language Macros	16
4 An Implicit Directory	16
5 The Current Directory	16
6 The Local Path Directories	17
7 The Global Path Directories	17
7 Escaping from the Normal Search	17
Chapter 3. Creating New Command Shells	19
1 Inherited Values	19
2 Creating an Interactive WShell	20
1 The Console Specification	20
2 Initial Script Files	20
3 Inplace Shells	21
4 Quiet Openings	21
5 Defining the Task Name	21
6 Initial Commands	21
7 The Startup File	21
8 Opening a WShell from Workbench	22
3 Terminating a WShell	22
4 Background Shells	22
1 The Background Character	23
2 Background Input Streams	23
3 Background Output Streams	23
4 Piped Background Shells	24
5 Defining a UserShell	24
1 Defining a Custom Shell	24
6 The SetExecute Utility	24
Chapter 4. Using the DisplayHandler	25
1 The Console Specification	25
2 The DHOpts Command	25
1 Command Arguments	25
2 Replacing Existing Devices	26
3 Device-Level Defaults	26
4 Console Option Keywords	27
1 Option Examples	32
2 Old-Style Attributes	33
5 Console Menus	33
1 The Menu Description File	33
2 Attaching a Menu	34
3 Deleting a Menu	34
6 Screen Jumping	34

Chapter 5. Line Editing and Command History	35
1 Line Editing	35
1 Editing Modes	36
2 Cursor Positioning	36
3 Deleting Characters	36
4 Save Buffer Operations	37
5 Miscellaneous Editing Features	38
6 Clipboard Operations	38
2 Session History	39
1 Flow Control	39
3 Command History	40
1 Retrieving Commands	40
2 Search Keys	40
3 Clearing the History Buffer	40
4 Command History Options	41
5 Sizing the History Buffer	41
6 Preloading the Command History	41
7 Swapping History Contexts	42
4 Keymapping	42
1 What is a Keymap?	43
2 Building a Keymap	43
3 Loading and Using a Keymap	44
4 Input Event Handlers	44
Chapter 6. Resident and Built-In Commands	45
1 Code Purity Requirements	46
2 Maintaining the Resident List	46
1 Auto-Loading Resident Modules	47
2 Alternate Names for Resident Commands	47
3 Deleting Resident Commands	47
4 Ignoring Checksum Failures	47
5 Listing the Resident Commands	48
3 More on Code Purity	49
4 The Built-In Commands	49
1 ALIAS	50
2 CD	50
3 ECHO	51
4 ELSE	51
5 ENDCLI	52
6 ENDIF	52
7 ENDSKIP	52
8 FAILAT	53
9 IF	53
10 JUMP	54
11 LAB	55
12 MOUNTED	55
13 POPCD	55
14 PROMPT	56
15 PUSHCD	56
16 QUIT	57

17 RECALL	57
18 RESI	57
19 REXX	58
20 SKIP	58
21 STACK	58
22 STDIN	59
23 SWAPCD	59
Chapter 7. Command Aliases	61
1 Maintaining the Alias List	61
2 Local and Global Aliases	62
3 Argument Substitution	62
4 Echoing Alias Commands	63
5 Listing Options	63
6 Cyclic Definitions	64
7 Command Abbreviations	64
Chapter 8. The Prompt String	65
1 Prompt Keywords	65
2 Prompt Programs	66
3 Variable and Backticked Command Expansion	66
4 Titlebar Prompts	67
1 Remote Titlebar Updates	67
Chapter 9. Command Piping	69
1 The PIP: Handler	69
2 Designing Programs for Piping	70
3 Example Filter Programs	70
1 A Pipe "Tee"	71
2 Head	71
4 The ExecIO Utility	71
1 Command Arguments	71
2 ExecIO Examples	72
Chapter 10. Using REXX-Language Macros	75
1 Activating the ARexx Server	76
2 Invoking REXX-language Macro Programs	76
3 The REXX Search Path	76
1 Early Search Termination	77
4 Capturing Output from Commands	77
5 Inline REXX Programs	78
1 Inline Aliases	79
2 "Resident" REXX Programs	79
6 Communicating with Other ARexx Hosts	80
7 Learning More About REXX	80

Chapter 11. The Configuration File	83
1 Configuration Records	83
2 Modifying the Configuration	83
1 The OPTIONS Record	84
2 The SPECIAL Record	85
3 The ALIAS Record	86
4 The RESI record	86
Chapter 12. Filename Completion	87
1 Filename Completion	87
1 The Completion Process	87
2 Parsing the Command Line	88
3 Selecting the Names	88
4 Displaying the Completions	89
5 RAW: Mode Operation	89
2 Command Line Options	90
3 The FComp Configuration File	91
1 The OPTIONS Record	91
2 The FILETYPE Record	91
3 The KEY Record	92
4 The COMMAND Record	92
4 Output Formats	93
5 Keymapping with FComp	94
6 WShell in the Workbench Environment	95
1 AppWindow TOOLTYPE Conventions	95
7 Caching and Memory Management	96
Chapter 13. Using the PathHandler	97
1 Defining a Path	97
2 Path Aliases	98
1 Implicit Paths	99
3 Protection Attributes	99
4 Directory Timestamping	100
5 Technical Details	100
Appendix A. DisplayHandler Control Sequences	103



Chapter 1

Introducing WShell 2.0

WShell 2.0 is an enhanced command shell designed as a highly-compatible replacement for the Amiga's Shell or Command Line Interface (CLI). It provides a much more powerful command environment while maintaining full compatibility with virtually all existing software. Among the many features supported by WShell are:

- Superior Line-Editing, Command History, and Session History
- Console-Window Menus
- Command Aliases and Abbreviations
- Resident and Built-In Commands
- An Implicit CD Command
- Extended Prompting and Window-Title Options
- Iconic (Drag-and-Drop) Operations with Workbench
- Filename Completion
- Fully Concurrent Piping
- REXX-Language Macros using ARexx

WShell is implemented as a shared library so that only one copy of the software is loaded into the computer's memory, thus minimizing the use of memory resources. Since it is fully compatible with the AmigaDOS CLI and AmigaShell, you can run all of your existing software with it. WShell 2.0 will operate under both AmigaDOS 1.3 and 2.0, although some features are available only with the 2.0 OS software.

WShell uses a proprietary console handler, called the DisplayHandler, to provide the improved line editing, command history, and session history features that make a command environment more productive and enjoyable. WShell will also work with the AmigaDOS CON: or NEWCON: console handlers, or with any other handler that supports the documented handler interface, but with a reduced set of capabilities. The ConMan software used by previous versions of WShell is not needed with WShell 2.0.

The WShell package also includes several handy auxiliary utilities including the PathHandler for extended path-searching features and ExecIO for enhanced ARexx capabilities.

1-1 Compatibility

One of the principal design goals of the WShell software was to provide complete compatibility with the Amiga's Shell and CLI. Hundreds of hours were spent studying AmigaDOS to learn the undocumented rules it uses to launch a program and provide the support environment. The end result is 100 percent compatibility with the AmigaDOS command programs—all of the standard command programs in your C: directory will work correctly with WShell.

Virtually all third-party and user-developed software should also work fine with WShell, with the possible exception of some of the **make** utility programs used for software development. Utilities like **make** are sometimes written to be highly specific to a particular command shell. However, you should be able to use such programs by preceding the command with a **run**, as in **run make**.

If you encounter other programs that work correctly with the standard CLI but not with WShell, please report them immediately so that the problems can be corrected.

1-2 Using this Manual

As one of the goals of WShell was to make the command environment easier and more enjoyable to use, this manual does not assume much prior familiarity with the Amiga's Shell or CLI. It therefore contains some material that can be skipped by those already familiar with the command environment.

- Chapter 1 has your attention at the moment and will shortly tell you how to install the WShell 2.0 software.
- Chapter 2 reviews the command-line environment and the conventions used in the shell.
- Chapter 3 describes the commands used to create new instances of WShell.
- Chapter 4 explains the operation of the DisplayHandler and describes its various options.
- Chapter 5 describes the DisplayHandler line editing, command history, and session history features.
- Chapter 6 describes the resident list facility and built-in commands.
- Chapter 7 presents the command alias and abbreviation features.
- Chapter 8 describes the extended prompt string and window title features.
- Chapter 9 explains *piping*, a facility that allows you to direct the output of one command to the input of the next.
- Chapter 10 discusses the ARexx interface used to support REXX-Language macro programs.
- Chapter 11 describes the WShell configuration file.
- Chapter 12 presents FComp, the WShell filename-completion utility.
- Chapter 13 describes the PathHandler, a pseudo-device that provides an enhanced path-searching capability.
- Appendix A includes a table of the escape-sequences used by the DisplayHandler.

Although much of the material here will be familiar to users of previous versions of WShell, many substantial changes were made during the implementation of WShell 2.0. A careful reading of the manual will be essential to using the new software to the fullest extent.

Typographic Conventions

This manual uses several typographic conventions to help clarify the presentation. All of the terms and words that might be entered as input to the computer, or received as output from the computer, will be set in typewriter font like this. The surrounding text will remain in the body font.

Command templates follow a standard convention to indicate choices and options. Optional arguments are shown enclosed in square brackets [like this], and alternate choices are { enclosed in brackets and separated | by a vertical bar }. Command keywords are in uppercase typewriter font.

1-3 Installing the WShell Software

WShell is implemented as a system of layered software, much along the lines of the Amiga's operating system. It has several different components which are loaded when the first WShell opens and usually stay resident until you reboot the computer. The WShell software should be installed on any of your system (bootable) disks from which you plan to use a command environment.

Installation consists of copying files from the distribution disk to your system disk, and optionally making changes to your system Startup-Sequence file. A command script to install WShell (called Install-WShell) has been provided, but in this section we'll briefly discuss the installation process. You'll probably want to study the Install-WShell file to understand the default installation process.

Library Files

Shared libraries contain functions designed to be called from any number of tasks simultaneously. The library concept is central to the design of the Amiga's Operating System, and it has been used for WShell as well.

Libraries normally reside in a special directory with the assigned name of LIBS:. This directory is usually assigned to your boot disk, but can be reassigned to another drive (for instance, a hard disk) during your system startup. WShell uses a library called wshell.library, and it must be copied from the :libs directory of the distribution disk to your system LIBS: directory.

When you first start a program that opens a disk-based library, the library file is loaded into memory and remains there until you reboot. It is possible for unused libraries to be purged from memory if you run low, however, so the LIBS: directory should remain accessible in case the library needs to be reloaded. Libraries can be purged only if no programs are using them at the time, so there's no danger of having one pulled out from under you.

In the event that it is inconvenient or impossible to copy a library file to the LIBS: directory, there is another way to load the library. The :c directory of the distribution disk includes a command called loadlib that can be used to load a library from an arbitrary directory, instead of just from LIBS:. You can use loadlib in your startup-sequence and load the libraries from whatever disk you've chosen to store them. For example, loadlib dh0:wshell.library would load the WShell library from the root of dh0:.

Utility Programs

Several of the WShell facilities have been implemented as executable utility programs that set up the required interfaces when run. These utilities include the DHOpts program, which initializes the DisplayHandler software, and FComp, the filename-completer. These programs are generally needed only when the computer is first booted and can be placed anywhere in the initial search path, or can be invoked with an explicit path. The default installation script copies these programs to the system C: directory, but you may wish to define a separate directory for them.

Handlers

The PathHandler software uses the file :l/pathhandler as its interface to AmigaDOS. This file should be copied to your L: directory if you plan to use the PathHandler.

To use the PathHandler you'll also need the mountlist entry path-mountlist. This file should be copied to your DEVS: directory, or alternatively can be added to your system DEVS:mountlist file using your preferred text editor.

Although not strictly a handler program, the :l/wshellseg file should be copied to your L: directory if you intend to use WShell as the "UserShell" under AmigaDOS 2.0. Refer to Chapter 3 for information on the UserShell facility.

Configuration Files

WShell can be easily customized by modifying its configuration file S:Config-WShell. For the initial installation you should probably use the example configuration from the distribution disk, so simply issue the command `copy :s/Config-WShell S:`. Chapter 11 describes the structure of the configuration file, should you wish to modify it to customize your WShell environment.

The FComp utility also uses a configuration file, not surprisingly called Config-FComp. It should be copied to your S: directory using the command `copy :s/Config-FComp S:`. Refer to Chapter 12 for a description of the FComp configuration options.

Command Files

Since WShell offers many features that aren't found in the standard Shell, it includes a few commands that work only with WShell. These commands are in the :c directory of the distribution disk and should be copied either to your system C: directory, or to another directory in your standard shell search path.

Once you've installed the WShell software and are satisfied that it is working properly, you may be able to delete some files from your C: directory. A number of the AmigaDOS 1.3 commands have been built into WShell and no longer need to remain on your system disk. In addition, you should no longer need to use the `newcli` command, as it is effectively replaced by the `newwsh` command.

The Startup Sequence

The `:s` directory of the distribution disk contains some sample scripts (command files) to help you customize your Startup-Sequence. The files `Startup1` and `Startup2` are examples of a two-part startup sequence that will give you some suggestions for integrating WShell into your Startup-Sequence file. Since every system is a little different, it would be better to study these and combine them with your existing startup rather than using them verbatim.

Under AmigaDOS 2.0 the standard Startup-Sequence file invokes a separate script called `User-Startup`, and you may wish to keep all WShell-related startup commands in this file. An example `user-startup` file is included in the `:s` directory.

Startup Scripts

WShell allows you to define a default script file to be run whenever a new shell opens. If you wish to use this option, you must name the file `WShell-Startup` and place it in your `S:` directory. A sample `WShell-Startup` is included in the `:s` directory. Note that previous versions of WShell used the name `Startup-WShell` instead of `WShell-Startup`; the name was changed to bring it more in line with the system default.

Environment Variables

Environment variables are globally-defined named quantities that can be used to tailor or modify various programs. WShell uses several environment variables to control options that can be changed at any time. Chapter 2 describes the operation of the various environment variables used by WShell.

The WShell distribution disk contains an environment directory that can be copied to the `ENV:` directory to initialize it. The directory is stored as `:s/env` of the distribution disk and contains the files `path`, `titlebar`, `echo`, and `shellwindow`. Under AmigaDOS 1.3 you'll need to create an `ENV:` directory, if you don't already have one, but under AmigaDOS 2.0 the system automatically sets up `ENV:` for you. If you're installing WShell on an AmigaDOS 2.0 system, just issue the command `copy :s/env ENVARC:` to place the environment variables in the system archive directory.

To activate the WShell environment variables under AmigaDOS 1.3, create an environment directory on `RAM:`, `VDO:`, or some other permanently-mounted device, and use the `assign` command to define it as the `ENV:` device. Then copy the `:s/env` directory to `ENV:`. For example,

```
R(0); 01:44:53> mkdir ram:env
R(0); 01:44:55> assign ENV: ram:env
R(0); 01:44:59> copy :s/env ENV:
```

would define and initialize `ram:env` as the `ENV:` device. WShell will automatically detect the presence of the environment handler and will adjust its internal options accordingly.

Activating the DisplayHandler

The DisplayHandler console handler software must be activated with the `DHOpts` command before it can be used by WShell. This in turn identifies the DisplayHandler device names to AmigaDOS (in lieu of an explicit mount command) and establishes a series of default values used by the DisplayHandler.

Once you've installed the `DHOpts` command, you can activate the DisplayHandler for one or more device names using, for example, the command `DHOpts CNC: PIP:.` The selected DisplayHandler device names will then appear on the list of AmigaDOS device names (as shown by the `assign devices` command, for example.) Note that `DHOpts` uses the `wshell.library` for its operation, so the library must be available at the time the command is run.

The `DHOpts` command accepts several arguments and option switches; these are explained in detail in Chapter 5. If necessary, `DHOpts` can be run several times to set up all of the required DisplayHandler devices.

The above example activates the DisplayHandler as a separate `CNC:` console handler device, but doesn't override the Amiga's built-in `CON:` device. If you wish to replace the system `CON:` handler, you must first remove the prior definition using the appropriate command: `assign remove CON:` under the AmigaDOS 1.3, or `assign dismount CON:` for AmigaDOS 2.0. `DHOpts` will issue an error message if you attempt to override an existing (non-DisplayHandler) device name.

Regardless of the chosen method of activation, for convenience you should place the necessary commands in your `S:startup-sequence` (or `S:User-Startup`) file.

Activating Filename-Completion

Filename-completion is performed by the `fcomp` utility program. After you've installed the executable and its configuration file, just issue the command `fcomp` to turn on the filename-completion features. Refer to Chapter 12 for more information on this important facility.

Activating the PathHandler

To use the PathHandler software you'll need to "mount" it to establish it as a DOS device. This can be done using the command

```
mount PATH: from DEVS:path-mountlist
```

after you've installed the `:1/pathhandler` and `path-mountlist` files as previously described. Refer to Chapter 13 for instructions on defining and using extended paths with the PathHandler.

1-4 Testing the WShell

After you've installed the required files you should verify that the WShell software is working correctly. From a Shell or CLI, first issue the command `DHOpts CNC:` to activate the DisplayHandler. Now issue the command `list >CNC:wait S:` and verify that the new window has a scrollbar (proportional gadget) in the right-hand border. You can use the

scrollbar to browse the session history, and the window should remain open until you enter a control-\ character. If the window fails to open or doesn't have a scrollbar, go back and recheck your installation.

After the DisplayHandler has been activated, open a WShell with the `newwsh CNC:` command. It should open a new window and display its copyright notice followed by a "smart" prompt. Try typing a few commands at the new shell, and play around with the arrow keys. You should be able to move about in the command line and recall previous commands. If the `newwsh` command fails to open a window or if the line editing features aren't working, go back and recheck your installation.

1-5 Using WShell from Icons

WShell can be activated and configured from Workbench using the various icons provided with the distribution disk. From Workbench you can open a WShell by double-clicking on the `NewWSH` icon. If you haven't placed a `DH0pts` command in your startup-sequence, you should double-click the `DH0pts` icon first. The DisplayHandler controls only those console windows opened after it has been activated.

The WShell icons can be customized using various *tooltype* strings, which are simply text strings stored within the icon file. The icons on the distribution disk contain examples of the supported tooltypes, with certain tooltypes deactivated by an asterisk preceding the tooltype name. If you prefer to operate WShell from Workbench, you can duplicate and modify the icons to customize your setup. From Workbench you can display and modify an icon's tooltypes by selecting the icon and then choosing the "Info" menu option. Don't forget to make appropriate changes to the filename paths for any files referred to by a tooltype string.

Note that if you haven't yet installed the WShell software, it may be necessary to first load the `wshell.library` file into memory using the `LoadLib` icon. For best results we recommend installing the software prior to use, but it is possible to test and use WShell from the icons before installation.



Chapter 2

The Command Environment and Shell Conventions

Most readers will be familiar with the Amiga's Workbench, an iconographic interface that lets you run software by just pointing and clicking on its "picture." Although the icon-based approach is undeniably easier to learn for novice users, many operations can be more succinctly stated in a traditional command language. Concepts such as conditional or repetitive execution of programs are impossible to express with the existing Workbench capabilities.

Fortunately, the Amiga is endowed with a parallel command-based environment that can operate concurrently with or as an alternative to Workbench. The standard command environment, called the *Command Line Interface* or CLI, is entered by double-clicking the CLI or Shell icon or by simply terminating your boot sequence before loading Workbench. In the CLI environment you give instructions to the computer in the form of *commands*, which are just text strings that describe what program to run and how you want to run it.

In this chapter we'll review the Amiga's command environment and command syntax, and examine the various command-line conventions used by WShell.

2-1 The Command Environment

When you enter a command line, the shell treats the first word of the command as the name of a program file to be executed. It searches for the program file, and if successful executes the program within a well-defined command environment.

The Current Directory

Within each shell a unique directory is identified as the *current directory*. This directory is used as the initial path for any filename that doesn't include an explicit device. The current directory is used both by the shell itself and by the programs executed from that shell. You can change or display the current directory using the CD command.

Since the commands and files in the current directory do not need to be prefaced with a full (absolute) path name, they can be referenced much more conveniently. For example, if you had a file with the full path name `df1:include/shell/shlib.i` and your current directory was `df1:include/shell`, then the file could be referred to as simply `shlib.i`.

Input and Output Streams

When the command shell runs a program, the program normally inherits the shell's input and output streams. These default input and output streams allow the program to read from and write to the shell's console window. The input and output streams can be changed using *redirection* with the command; refer to the following section for further information on redirection.

Stack Size

Prior to running each command, the command shell sets up a memory area called the *stack* for the command to use. It's important that every program have enough stack for its execution requirements, but unfortunately there is no system convention to allow a program to specify its stack requirements. The standard AmigaDOS commands and most third-party applications generally work well with the default 4,000 byte stack, but some programs may require substantially more. You should check the documentation of your software packages for more information on stack requirements.

The stack size can be adjusted using the built-in `stack` command. Refer to Chapter 6 for information on this command. WShell also supports an option to read the required stack from a program's icon file, if it has one. This can be enabled using the `CHECKICON` keyword in the configuration file; refer to Chapter 11 for more information.

PROGDIR: Program Directory

Under AmigaDOS 2.0 an executing program can determine the directory from which its program file was loaded. This directory is referred to using the "PROGDIR:" logical name. The shell is responsible for setting PROGDIR: to the appropriate directory after it finds an executable file.

2-2 Command Structure

Commands have a very simple verb-and-object structure in which the verb is the name of a program to execute, and the objects are file names or modifier switches that direct the program. The general form of a command can be expressed as

program [redirection] [arguments] [piping-specifier command] [background-specifier]

in which each part of the command line prototype has a specific format. These parts are described below.

Program Names

The first word of the command line usually specifies the file name of an executable or REXX macro program. File names are not case-sensitive and can include *device names* like `df1:` and *directory names* separated by a `/`. For example, `list`, `C:list`, and `df0:system/setmap` are all valid program names.

The first word of the command may also refer to an entry on the local or global *alias list*, in which case the associated alias value is substituted for the original command word. Aliases are explained in detail in Chapter 7.

Redirection Specifiers

Redirection allows you to specify a file as the input or output stream for the program being executed. The angle-bracket characters `<` and `>` are used to indicate redirection and must be preceded by a space to be recognized as such. WShell accepts four forms of redirection:

- *<filename* indicates redirected input,
- *>filename* indicates redirected output,
- *>>filename* indicates redirected output appended to an existing file, and
- *<>filename* redirects both input and output streams to a console device.

The file name for redirection follows the same model as for a program name.

The file specified for input redirection must exist before the command can be executed. Output redirection normally creates or overwrites its target file as necessary, and it's therefore possible to destroy an important file by accidentally specifying it as an output target. WShell allows you to define a `NOCLOBBER` local or environment variable to prevent this from occurring: when `NOCLOBBER` is in effect, output redirection will not overwrite an existing file, and append redirection will not automatically create the target file. If necessary, you can override this protection mechanism by appending an exclamation mark to the redirection specifier, as in `>!` or `>>!`.

Note that under AmigaDOS 1.3 the standard shell (or CLI) requires that you enter all redirection specifiers before any command arguments. WShell has no such restriction, so you can place redirection specifications anywhere after the command name.

Once the first occurrence of a redirection specifier has been recognized, WShell ignores any subsequent occurrences of the same specifier. In some cases you may need to use a dummy specification of the form `<*` or `>*` to postpone the recognition of redirection. For example, to redirect the output of a command being run, rather than the `run` command itself, you could enter `run >* myprogram >myfile`.

Command Arguments

The command line can include whatever additional data are required by the command program. WShell does not interpret these argument fields, but simply passes them through to the command.

Piping Specifiers

Piping is a mechanism that allows you to direct the output of one program as the input to another, and is actually a form of combined output/input redirection. Piping is indicated by the `|` character followed by another entire command line (which itself could include piping as well.) Piped command lines are processed by spawning additional WShells to process the remaining command line. Piping is an important capability and is explained in greater depth in Chapter 9.

Background Specifiers

WShell supports a *background character* to indicate that a command is to be run in the background rather than executed directly. By default the background character is an ampersand (`&`). When WShell encounters a background character in a command line, it spawns a new shell to process the command up to that point, and then continues processing the remainder of the command. The default background character, and the option to accept or reject multiple background characters, can be changed in the WShell configuration file. Refer to Chapter 11 for more information.

2-3 Comments, Quoting, and Escape Characters

Several other characters have special meanings in a command line.

Quote Character

AmigaDOS commands generally expect command line arguments in the form of words separated by spaces. Occasionally you may need to consider several words as a single argument, and for these cases you must enclose the argument in double-quotes. When the AmigaDOS command-line parser encounters a double-quote, it accepts everything up to the closing double-quote as a single argument. For example, "My Funny Name" would be accepted as a single argument.

Escape Character

Within a quoted field WShell recognizes the asterisk *** as an *escape character*, and any character following an asterisk is accepted as a literal. For example, you can use this escape mechanism to include a double-quote character within the quoted string. The escape character can be changed by specifying an alternate character in the WShell configuration file; refer to Chapter 11 for more information.

The AmigaDOS commands recognize two special characters following an escape: the sequences **E* and **N* are translated to ASCII ESC (hex 1B) and line feed (hex 0A) characters, respectively. You can use these to include a line feed ("newline") in a message or prompt string, for example. Note that this interpretation is made by the command programs and not by the shell; the shell simply passes the argument string to the command.

Special Escapes. The escape character is normally active only within a quoted argument, but can also be used to escape certain special characters recognized by WShell. If any of the characters *<*, *>*, *|*, *&*, *[*, *‘*, *;*, or the *** escape character itself, are preceded by an escape character, the escape character is removed and the following character accepted as literal text. This "special-escape" mode must be enabled with the SPECARG configuration option; refer to Chapter 11 for more information.

Comments

On a command line the semicolon (*;*) is a *comment* character, and anything that follows the semicolon is ignored. Comments are usually used only in command files, but can be entered interactively as well.

If you need to include a semicolon in a command line, you must enclose the command in double-quotes. For example, you could enter an inline REXX program as

```
"trace r;do i=101 to 105;'fault' i;end".
```

Continuations

WShell supports a mechanism to allow you to enter several command lines at once. If you enter a line ending with a *+* character, the shell replaces the *+* with a "newline" and reads another line of input. There is no fixed limit to the number of lines you can enter this way. This form of command continuation is normally used only with the *runwsh* command.

2-4 Variable Expansion and Backticked Commands

WShell supports two forms of command-line text substitution to give you greater flexibility in creating and processing commands. *Variable Expansion* allows the command line to include the names of environment or local variables whose values are to be substituted when the command is processed. *Backticked Commands* are subcommands embedded in the main command whose output text replaces the original subcommand text.

Both the variable expansion and backticked command processing are completed before the command is scanned for final execution, so the model command structure previously described remains valid. If the WShell echo facility is enabled, the expanded commands will be displayed before execution, allowing you to verify that the desired substitutions were performed.

Variable Expansion

Variables to be expanded in a command line are indicated by a preceding \$ character. The variable name is taken as the text following the \$ up to the next delimiter character. Variables names normally consist of just alphabetic (A-Z, a-z) and digit (0-9) characters, but other characters can be included in the name by surrounding it with curly braces {}. For example, \$MyName1 and \${A.Funny //Name} are valid variable names.

Variables are expanded by reading the value, if any, of the designated variable and substituting the text in place of the variable name. If the variable doesn't exist, the original text is left unchanged.

Variable expansion is performed in one complete pass through the command line and precedes the expansion of backticked commands, if any. The variable values may therefore include partial or complete backticked commands. For example, if the variable partial contained the text date, then the command echo Date '\$partial' would be expanded to echo Date 'date' after the first pass.

Backticked Commands

Backticked commands are subcommands embedded in a larger command and delimited by backtick (') characters. The subcommands are processed by executing them as commands in a subshell. The text output of each subcommand is scanned to replace any "newline" characters with spaces and then substituted in place of the original command text. Backticked subcommands are processed from left to right.

Backticked commands can include other backticked commands by using a doubled backtick for the nested command. For example, echo 'dir ''list quick ram:'' opt a' includes the command dir 'list quick ram:' opt a, which itself includes list quick ram:. Each embedded command is run before the command containing it, and eventually the original command is executed with the appropriate substituted text.

The expansion of backticked commands occurs in the second of two passes through the command line, after all variables have been expanded.

If an error occurs in a backticked command, WShell will echo the substituted text (if any), but will abort the original command. This allows you to determine the probable cause of the error, but protects against the possibly dangerous execution of an incorrect command.

Note that WShell uses a more general model for backticked commands than that supported by the AmigaDOS 2.0 Shell. In particular, the AmigaDOS Shell will accept only a single backticked command on a command line, and does not support nested backticked commands. It is therefore possible to create script files using the WShell features that will not work properly with the plain AmigaDOS Shell.

2-5 Environment and Local Variables

WShell uses several environment or local variables to control various aspects of its operation. These variables are shown in the table below.

Table 2.1 Environment Variables

Variable	Action	Example
Echo	Controls Echoing of Commands	ON
NoClobber	Prevents Overwrite During Redirection	ON
Path	Global Path Directories	df1:c,C:
Titlebar	Window Title String	Process %n >>>%c
Shellwindow	Default Console Specification	CON://640/100/New/c

If defined, a local variable will override the corresponding environment variable, thus allowing you to use local variables to tailor the behavior of an individual WShell. Local variables are supported only under AmigaDOS 2.0.

The distribution disk contains an environment directory :s/env with examples of the values used with the variables. Refer to Chapter 1 for information on installing these values.

Local and environment variables can be modified under AmigaDOS 2.0 using the set and setenv commands, respectively. AmigaDOS 1.3 does not support local variables, but environment variables can be set by simply redirecting the echo command to the desired ENV: file. For example, echo >ENV:path C: would set the ENV:path variable to C:.

2-6 The Search Path

Every time you enter a command the shell must search for a program to execute. Understanding this search process is essential to using WShell effectively.

The novice user might ask why such a search is even required—shouldn't WShell automatically know how to perform each command? The answer is that the shell is really just a tool to allow you to interact with the computer, and therefore can't anticipate every command that you might someday want to execute. Instead, it allows you to specify the way that the shell should search for a command.

WShell looks for a command by following a prescribed search path until it locates a file matching the command name. The full search hierarchy followed by WShell is

- The Previous Command
- Built-In and Resident Commands
- REXX-Language Macro Programs
- An Implicit Directory
- The Current Directory
- The Local Path Directories
- The Global Path Directories.

The last three stages of the search—the various path directories—may be searched more than once. WShell tries to avoid unnecessary requestors by checking whether the search directory is on a mounted volume. On its first pass WShell looks only at those directories that are currently mounted. If the command hasn't been found and at least one directory was unmounted, the second search pass will request that you mount the necessary volumes. The second pass terminates immediately if one of the requested volumes cannot be mounted.

If WShell finds a file matching the command name in one of the search directories, it will try to load it unless the *execute protection* flag for the file indicates that the file is not executable. Thus if the file status (usually shown as *rwed*) doesn't include the *e* attribute, WShell won't attempt execute it. This test is more strict than that used by the standard CLI in the AmigaDOS 1.3 OS, but the AmigaDOS 2.0 Shell also requires that the execute bit be set.

If WShell can't locate your command, it issues the message

Unknown command *program-name*

and then returns (to the prompt or script file) with the return code set to 20. If the shell was executing a script file, the script will terminate unless the failure level has been set greater than 20.

The Previous Command

Under certain conditions WShell will *cache* (retain) an executable program after it exits, on the assumption that you may need to run it again. If the current command name matches that of the previously-loaded command, the previous command is re-executed. The previous command is cached for possible reuse only if it passes the same strict test for code purity used for resident commands. Refer to Chapter 6 for more information on code purity.

Command caching is controlled by the *CACHELIM* configuration option; refer to Chapter 11 for more information.

Resident and Built-In Commands

Resident commands are executable programs that have been loaded into memory and placed on the *resident list*. WShell then searches this resident list for a name matching the current command name.

Under AmigaDOS 2.0 WShell actually searches two resident lists: the private list maintained by WShell as well as the AmigaDOS (public) resident list. The WShell resident list is searched first, so that a command placed there will have precedence over a like-named command on the AmigaDOS resident list.

In addition to the resident commands, WShell provides a number of built-in commands that are part of the shell program itself and are always available. Built-in commands are automatically placed on the WShell resident list and are searched as part of the resident list search. The built-in commands are described in Chapter 6.

You can use the `resi` command to display the current resident commands. Refer to Chapter 6 for more information on resident and built-in commands.

REXX-Language Macros

The search for a REXX macro program occurs only if the ARexx server process is currently active. If it is, WShell sends the command to the ARexx server, which then conducts its own search for a macro program. REXX macro programs have their own search path maintained by the ARexx server. Refer to Chapter 10 for more information on using REXX macros.

An Implicit Directory

If the command name refers to a subdirectory of the current directory, or is the full path specification of a directory, WShell treats the command as an *implicit* `cd` and sets the current directory to the new directory. This provides a fast and convenient way to move from one drive to another or from level to level within a hierarchy. The usual directory shorthand notation is accepted, so that `/` moves up one level and `:` goes to the root directory.

WShell won't check for an implicit directory if the command line includes arguments or redirection, as these could not be used meaningfully with a directory. Thus if you wish to run a command with the same name as a directory, you can append a `>*` to disable the implicit directory option.

Normally the test for an implicit `cd` precedes the search for a disk-based command, so that a directory name has precedence over a like-named executable file. The `LATECD` configuration option allows you to postpone an implicit `cd` until the entire path has been searched. Refer to Chapter 11 for further information.

The check for an implicit `cd` can be completely suppressed with the `NOIMPCD` configuration option.

The Current Directory

Each shell has a unique directory identified as its *current directory*, and commands in this directory can be referenced without supplying a specific path. The shell searches the current directory for the command before searching any of the local or global path directories.

The Current Directory

Each shell has a unique directory identified as its *current directory*, and commands in this directory can be referenced without supplying a specific path. The shell searches the current directory for the command before searching any of the local or global path directories.

The Local Path Directories

The *local path* is a list of directories private to each instance of WShell. It is maintained exactly as is the path for the standard Shell and CLI, and the AmigaDOS `path` command can be used to add or delete path directories. The local path is searched sequentially until the program file is located.

Each directory in the local path is specified by a *lock*, a data structure that uniquely identifies a directory on a specific volume. This means that adding a path directory as `df1:c`, for example, does not specify the `:c` directory of whatever volume happens to be mounted in drive `DF1:`, but rather the volume that was mounted at the time the path directory was added. This is an important point to note, since if you have an unmounted volume in your local directory path, you may get frequent requests to mount the volume so that WShell can search it.

The Global Path Directories

The final search step examines each of the directories defined in the *global path*. These directories are defined as a name rather than as a lock, so a global path directory of `df1:c` will search whatever volume is currently mounted in drive 1. This complements the use of the local path directories and can be used to advantage if you must frequently swap disk volumes. If you want to specify a particular volume, you should use the volume name in the directory definition. For example, `WShell:c` would specify the `:c` directory of the WShell distribution disk.

The global path is maintained as an environment variable in the logical file `ENV:path` or as a path local variable. The path string is cached internally by the shell and is updated after every command. Individual path directories in the string can be separated by a comma, semicolon, vertical bar (`|`), or by “white space”—spaces, tabs, or “newline” characters. You can display the global path by just typing it, as in `type env:path`.

If you haven't defined the path local or environment variable, WShell will search the `C:` directory, in keeping with the standard Shell and CLI. If you do define a global path, you must explicitly include `C:` in it if you want it to be searched.

2-7 Escaping from the Normal Search

On occasion you may need to defeat the normal WShell search process in order to run a particular command. For example, if the desired command has the same name as a resident command, the shell will always run the resident version. In order to “escape” from the search sequence, you can preface the command with a left-bracket (`[`) character. The left bracket is stripped from the command and the search proceeds directly to the implicit directory step. This avoids the search for resident commands, built-in commands, and REXX macro programs.

The “search-escape” character should be required only rarely. If you find yourself using it frequently, you may want to reconsider your choice of resident commands and path directories. Remember that you can always include an explicit path specification in a command name so that the normal search process will lead unambiguously to the desired command. For example, the command

```
df0:system/format drive df1:
```

would load and execute the `format` program from the specified directory—unless you had defined a very pathological alias for it!

The search-escape character can be changed with the `SEARCH` option in the WShell configuration file. Refer to Chapter 11 for more information.

Creating New Command Shells

On a multitasking computer like the Amiga it's often convenient to have several WShells available. You may be working on more than one project and want to have a separate environment for each one, or you may simply be running several programs at a time. Each WShell runs as a separate task and has its own set of associated resources. The initial environment for each shell is inherited from its parent task and may be subsequently modified by a command file or by interactive commands. It is important to understand that each WShell task is distinct from the others, and changing an attribute like the default stack size affects only that particular shell.

There are two types of WShell tasks, which are generally referred to as *interactive* and *background*. Both types of shell are run by the same software, but make different assumptions about the source of input commands. An interactive shell assumes that a "live" user is available to provide additional input, and it will prompt you whenever it is ready to process another command.

Background shells have a fixed set of input commands determined at the time the shell was created. After these commands (which are equivalent to a script file) have been processed, the shell exits quietly without prompting for more input. The name "background" is somewhat of a misnomer, as it implies a lower-priority task, which is not necessarily the case. Background shells begin executing at priority 0, but this may be increased or decreased by subsequent commands.

3-1 Inherited Values

Each new WShell attempts to recreate the command environment of its parent task. It does this by carefully examining the parent environment and then duplicating the values that it finds there. For some fields this is just a matter of installing a value, but other environment attributes (like the local path directories) require that the shell create a number of linked data structures.

All of this happens automatically, so you don't have to do anything about it; however, the material is included here to help you understand how the command shell works. The inherited attributes are:

- Current Directory
- Current ("*") Console Handler
- Current Directory Name
- Prompt String
- Stack Size
- Local Path Directories
- Local Aliases

- Local Variables (AmigaDOS 2.0 only)

Which values are actually inherited depends on how the new WShell was created. This is because the existence of some of the inherited fields depends on whether the parent (the task creating the WShell) is a *task*, a *process*, or a *CLI (command shell) process*.

In most cases the new WShell will be launched from an existing shell and will therefore inherit the full complement of values. However, shells launched from some background “hot-key” utilities may not be able to inherit a search path and stack size, or will inherit the values in effect when the hot-key utility was run. If you need initial values for these attributes, you may need to supply them by executing commands in the shell startup file (S:WShell-Startup by default.)

3-2 Creating an Interactive WShell

You can create an interactive WShell by using the `newwsh` command. The command first opens a new window (if a console was specified as the input source) and then creates a new WShell task to manage the window. As soon as the new WShell has initialized itself, it will prompt you for a command. The `newwsh` command is analogous to the AmigaDOS `newcli` or `newshell` commands, but accepts a more general set of command line arguments. The command template for `newwsh` is

```
CONSOLE, FROM, INPLACE/S, QUIET/S, NAME/K, COMMAND/K, CMD/K/F.
```

The Console Specification

The `CONSOLE` argument is usually a DisplayHandler `CON:` specification, but in general can specify any DOS device. Console windows are specified by their location and size in screen pixels in the form `CON:left/top/width/height/title/options`, where *left* is the distance from the left border, *top* is the distance from the top of the screen, *width* is the total width of the window, and *height* is its total height. The window *title* is optional, but the slash following the height must be given. If the window title includes any blanks or special characters, the entire console definition must be enclosed in double-quotes.

The *options* field supplies a series of keyword arguments separated by slash (/) characters that allow you to customize the shell’s console window. For example, you could request a close gadget (/CLOSE) for the window, or perhaps move the scrollbar to the lefthand border (/LEFT). The DisplayHandler offers an extensive set of keyword options, which are described in Chapter 4.

If no `CONSOLE` argument is specified, WShell checks for a `ShellWindow` environment variable and uses its contents as the console definition. If no `ShellWindow` variable has been defined, the default console definition `CON://640/100/New WShell/c` is used.

Initial Script Files

If you want to run a command file as the first action of the new shell, you can specify a file following the (optional) `FROM` keyword. The shell will process this file and then return to interactive mode. The command file used in this context cannot include any parameter substitutions (this same restriction applies to the `newcli` command.)

Inplace Shells

In some situations you may wish to have a new WShell “take over” an existing shell, adopting its window and any partially-completed batch commands. This can be accomplished using the **newwsh INPLACE** command. If no console argument is specified, the existing shell window is used.

INPLACE was intended primarily for use in the startup sequence, but may be useful in other cases as well.

Quiet Openings

A new WShell usually announces itself with a brief message giving the software version and task number. In some cases this message may cause unwanted side effects—for example, by forcing an auto-open window to open prematurely. The **QUIET** switch will suppress this message.

Defining the Task Name

By default a new WShell task will be named **New.WShell**, but you can specify an alternate name with the **NAME** keyword. For example, **newwsh name MyShell** would assign the name **MyShell** to the new task.

Alternate task names are not often required, but may be convenient if you need to identify a particular shell to other software in the system.

Initial Commands

The **newwsh** command will accept an arbitrary command line as its initial command if either the **CMD** or **COMMAND** keyword is specified. The command following the keyword is processed just as though it had been entered at the prompt. The command line could therefore be an **execute** command to allow parameter substitution in the command file, or could specify a REXX-language macro program. Only one of **CMD** or **COMMAND** should be specified, and a command argument will override a **FROM** specification, if present.

For many commands either of the **CMD** or **COMMAND** keywords can be used, but there is a subtle difference between the two keywords. **CMD** accepts the remainder of the command line as a literal argument without checking for and extracting double-quoted fields. This allows you to conveniently pass a command line containing double-quotes without having to escape them, but prevents you from using ***E** and ***N** inside a quoted field to generate those special characters. In contrast, **COMMAND** extracts its argument using the AmigaDOS convention, so that ***E** and ***N** can be used in a quoted string, but any literal double-quotes must be escaped.

The Startup File

If neither a **FROM** file nor an initial command are given, WShell will look for a default startup file named **S:WShell-Startup**. If such a file exists, it will be processed just as though it followed the **FROM** keyword. The startup file allows you to specify a default set of commands to be executed whenever a new shell opens.

Opening a WShell from Workbench

If you're operating from Workbench, you can open a new WShell by double-clicking the **NewWSH** icon. The icon file lets you specify the console, **FROM** file, **INPLACE** switch, **QUIET** switch, task name, or command line with the **CONSOLE**, **FROM**, **INPLACE**, **QUIET**, **NAME**, and **CMD** or **COMMAND** tooltypes, respectively. To change the default tooltypes, pull down the Workbench **Project** menu and select **Info**. The **Info** tool will open its window and let you inspect or modify the various tooltypes. Since only one of the **FROM** or **CMD** tooltypes can be chosen, you should either delete the unwanted one or "comment it out" by preceding it with an asterisk, as in ***FROM**.

If you haven't yet installed WShell on your system, you should first double-click the **LoadLib** and **DHOpts** icons before attempting to activate WShell. The icons are provided as a convenience, but you should follow the installation procedure outlined in Chapter 1 to use WShell most effectively.

3-3 Terminating a WShell

You can terminate an interactive WShell by using the standard **endcli** command, but there is a more convenient way to close the shell. If you've attached a close gadget to the window (by including the **/AUTO** or **/CLOSE** option in the console definition), just hit the close gadget and the WShell will close. The same effect occurs if you press the **CTRL-** key five times in succession. Each **CTRL-** generates an "end-of-file" (EOF), and after receiving five of them the WShell assumes that you want it to close. It does warn you of the impending end, however, by replacing your normal prompt with one like **Ending 3>** after the second **CTRL-** is received.

The number of consecutive EOFs required to close the shell can be set using the **EOFLIM** configuration option. Refer to Chapter 11 for more information.

A word of caution concerning the close gadget—when you first start using this feature it's easy to get carried away and close down all of your command shells. If you've loaded Workbench or have installed a background "instant CLI" program, you can easily open a new WShell. Otherwise, you'll have to reboot the computer.

3-4 Background Shells

You can create a background shell by using the **runwsh** command, which is analogous to the AmigaDOS **run** command. The shell will process the command supplied with the **runwsh** command and then exit.

If you want to run several commands with a background shell, the commands can be strung together by ending each line with a **+** character. For example, the sequence

```
1> runwsh cd df1:+  
list quick
```

would first execute the **cd** command and then execute the **list quick** command. As always, the WShell exits after exhausting its input stream.

The `runwsh` command can also be used to read commands from its input stream, allowing you to pipe commands to be run by the background shell. For example,

```
list lformat "list %s" | runwsh
```

will create a series of `list` commands and run them in a background shell.

The Background Character

WShell interprets the ampersand `&` as the *background character*, a shorthand notation to specify running a background command. If you append an ampersand to a command, WShell runs the command in the background just as though it had been preceded by a `runwsh` command. For example, the command `myprogram &` is equivalent to `runwsh myprogram`.

You can place several background characters in a command line to run multiple background programs, as in the command `firstpgm & secondpgm & anotherpgm &`. However, if you would rather not have WShell interpret the `&` character within a command line, you can specify the `ONEBACK` option in the WShell configuration file. This limits the use of the background character to just the end of the line. Refer to Chapter 11 for more information.

The configuration file provides a `RUNBACK` option to change the background character to something other than the default ampersand. Changing it to a space character will effectively disable the background option.

Background Input Streams

The standard AmigaDOS CLI has a limitation that occasionally causes problems with some commands. Instead of supplying an interactive input stream as the default for a background command, as the command would get if it were run interactively, the `run` command supplies a `NIL:` stream as the standard input. Commands that prompt for user input from the standard stream may either resume execution immediately or hang forever, depending on whether they expect arbitrary input or a specific response.

Rather than perpetuate this problem, the `runwsh` command will supply an interactive input stream to the program. In general, this stream is shared with the parent shell, so the two shells obtain input alternately, but at least this provides a mechanism to supply input to the background program. If you want to run the command with a `NIL:` input stream, just use redirected input as in `df0:system/format <NIL: drive df1: name New`.

Background Output Streams

The output stream for a background shell is normally the same console as the parent. The background shell obtains an additional filehandle for the console window to ensure that it can't be closed suddenly. This means that even after the parent shell is terminated, the console window won't actually close until all background shells have ended. You may occasionally find that a shell refuses to close its window after terminating due to a remaining background task.

If you wish to cut off a background command so that its parent window can be closed, you can specify output redirection to `NIL:` with the `runwsh` command. For example, the command `runwsh >NIL: myprogram` will run `myprogram` in the background without holding

the original shell's window open, unless the program itself opens an additional filehandle on the window.

It is also possible to cut off a background command launched by the background character `&`. In this case you must specify both input and output redirection to `NIL:`, as in `myprogram <NIL: >NIL: &`.

Piped Background Shells

An implicit background shell is created whenever you use *command piping*, which is described in Chapter 9. Piped shells behave like other background shells, except that they run at the same task priority as their parent.

3-5 Defining a UserShell

AmigaDOS 2.0 provides a mechanism by which an external shell program can be installed as the default shell for many system operations. This selected shell is called the UserShell, and WShell follows the conventions required for UserShell operation.

To declare WShell as the UserShell, you must ensure that the `wshellseg` file is in your `L:` directory, and then issue the command

```
resident Shell L:wshellseg SYSTEM PURE
```

to add the `wshellseg` segment to the resident list.

Once installed as the UserShell, the system `newshell` and `run` commands will invoke WShell, and the Workbench Amiga-E command execution will be processed by WShell. The AmigaDOS 2.0 `System()` function will also invoke the UserShell if the `SYS_USERSHELL` tag is specified in the calling sequence.

Defining a Custom Shell

WShell can also be installed as a system *custom shell*, in which case it will be invoked by any call to `System()` requesting that shell name. To install WShell as a custom shell, verify that the `wshellseg` file is in your `L:` directory and issue the command

```
resident name L:wshellseg SYSTEM PURE
```

where *name* is the desired shell name.

3-6 The SetExecute Utility

AmigaDOS provides a function called `Execute()` to run an arbitrary command in a shell environment. Usually this function invokes the system shell to run the command, but the `SetExecute` program provided with WShell patches the DOS function to use WShell instead. This allows any commands issued by a program via `Execute()` to be run in a WShell environment, so that the commands are processed using the WShell alias and resident lists, and will be passed to ARexx if appropriate.

There are possible side effects to installing the `SetExecute`, as the WShell environment generally has a different set of aliases and resident commands and a different command search order. Usually these problems can be resolved by checking the actual command strings issued by the particular application and making any necessary changes.

Chapter 4

Using the DisplayHandler

The WShell software includes a custom console handler called the DisplayHandler that provides line editing, command history, session history, and window control features. Because of its many features and options, we'll devote two chapters to the DisplayHandler. This chapter covers the commands and options used to define and open a DisplayHandler console, and Chapter 5 describes the various editing and control features of the handler.

4-1 The Console Specification

The DisplayHandler follows the AmigaDOS conventions for specifying a console window. Console windows are specified by their location and size in screen pixels in the form

`CON: left/top/width/height/title/options`

where *left* is the distance from the left border, *top* is the distance from the top of the screen, *width* is the total width of the window, and *height* is its total height. The window *title* is optional, but the slash following the height must be given.

The *options* field supplies a series of keyword arguments separated by slash (/) characters. The DisplayHandler provides an extensive set of options to allow you to customize the console window attributes and handler operation; these are described in a later section.

4-2 The DH0pts Command

The DisplayHandler uses a special command called DH0pts to mount the handler as one or more DOS device names. In addition to identifying the handler to AmigaDOS, DH0pts maintains a set of default values for the DisplayHandler devices and manages the menu descriptions used with the DisplayHandler.

The DH0pts command line can specify up to five mount names, as well as several options and the menu file. The command can be run several times if more devices are needed. DH0pts can be run from Workbench and may be included in the WBStartup drawer.

Command Arguments

The argument template for the DH0pts command is

`MOUNT, , , , RAW/S, STACK/K, MENU/K, NAME/K, REPLACE/S, DELETE/K`

and the template keywords are described below.

MOUNT Arguments. The MOUNT arguments (up to five) provide the names of the devices to be defined. The device names must include a colon :, and may be followed by an optional set of defaults for that device.

RAW Switch. The RAW switch specifies that the device should open in RAW mode.

STACK Keyword. The **STACK** argument provides sets the stack size for the handler. The default stack should be sufficient for most operations.

MENU Keyword. The **MENU** keyword specifies the name of a menu description file to be processed. Unless a **NAME** argument is provided, the tail part of the menu file name will be used as the menu's name. DisplayHandler menus are described in a later section.

NAME Keyword. The **NAME** argument supplies a name for the menu.

REPLACE Switch. The **REPLACE** switch tells **DHOpts** to update a menu description, if it already exists. By default **DHOpts** will not modify a menu description after it has been created.

DELETE Switch. The **DELETE** switch instructs **DHOpts** to delete the specified menu.

The device names chosen for the DisplayHandler are arbitrary except for the following special cases:

Special DisplayHandler Device Names	
Name	Operating Mode
PIP:	Pipe-mode Handler
CNN:	Non-Blocking Display
RAW:	Raw Mode
CNX:	Extensible Handler (default serial.device)

All other names will behave as a **CON:** device.

Examples:

```
1> DHOpts CNC: PIP: MENU S:Display/CNC-Menus
1> DHOpts MENU S:Display/CNC-Menus NAME MyMenu REPLACE
```

Replacing Existing Devices

DHOpts will not allow existing device names to be overridden; these must be dismounted explicitly if you wish to redefine a device. For example, to redefine **CON:**, you must first issue the appropriate command

```
1> assign CON: remove ; under 1.3 OS
1> assign CON: dismount ; under 2.0 OS
```

and then run the **DHOpts** program. **DHOpts** ignores requests to mount devices already assigned to the DisplayHandler.

4-3 Device-Level Defaults

The DisplayHandler allows you to define a set of default attributes for any DisplayHandler device. These defaults can supply a value for the window size and keyword options, which are then used unless explicitly overridden by a device specification.

Device-level default strings are specified as part of the device name on the **DHOpts** command line. **DHOpts** saves the text following the ':' separator in the name as the default options string for that device, and any console opened with the same device name will

inherit these default options. The defaults may be changed at any time by running the **DHOpts** command again.

The default string is processed just as a device name would be and will set values for any of the keyword options specified. The actual device name is processed after the default and may override the default option settings.

Example:

```
1> DHOpts CNC:0/11/640/100/New/CLOSE ; define CNC: defaults
```

4-4 Console Option Keywords

The DisplayHandler is highly configurable by means of keyword options in the console specification. It supports the AmigaDOS 2.0 **CON:** options in a compatible manner, but provides many more capabilities.

All of the DisplayHandler options can be specified by keywords separated by slashes (/) following the device name and window size, if present. Certain keywords expect one or more parameters following the keyword, which may be preceded by an optional space or # symbol. Keywords are case-insensitive and may appear in any order, except that the four window dimensions and window title must appear together as a group. Any unrecognized words are ignored.

Table 4.1 DisplayHandler Option Keywords

Keyword	Parameters	Description
ALT	4	Alternate size (e.g. ALT#0,10,640,200)
AUTO	none	Auto open and close (implies close gadget)
BACKDROP	none	Backdrop window
BPEN	1	Block pen color
BRKMODE	1	Break-signal control flags (hex)
CLOSE	none	Close gadget
COLS	1	Display columns (character width)
CUSTOM	1	Custom screen address (hex)
DEACT	1	Delay before deactivation (seconds)
DEVICE	1	Device name (CNX: only)
DPEN	1	Detail pen color
FONT	1	Font name (e.g. topaz.font)
FONTSIZE	1	Font size (e.g. 11)
HLINES	1	History buffer entries (0 for no command history)
INACTIVE	none	Open non-activated
LEFT	none	Left-border prop gadget
LIMIT	1	Pipe capacity limit (bytes)
KEYMAP	1	Keymap name
MENU	1	Menu name
MINSAVE	1	Minimum length to save in history

Table 4.1 (cont.) DisplayHandler Option Keywords

Keyword	Parameters	Description
NOALT	none	No alternate size
NOBORDER	none	Borderless window
NOCLOSE	none	No close gadget
NODEPTH	none	No depth gadget
NODRAG	none	No drag gadget
NONBLOCK	none	Non-blocking mode
NOPROP	none	No proportional gadget
NOSIZE	none	No size gadget
OVER	none	Overstrike mode
POPUP	none	Screen-to-front on opening
RAW	none	Open in RAW mode
RESAVE	none	Resave unmodified history lines
ROWS	1	Display rows (character lines)
S*	none	Open on front screen
SCREEN	1	Public screen name, or "*" for front screen
SESSION	1	Session buffer size (0 for no session history)
SHARED	none	Serial shared mode (CNX: only)
SIMPLE	none	Simple-refresh window
SMART	none	Smart-refresh window
SPILL	1	Spill file name
STICKY	none	Retain input mode after RETURN
TRIES	1	Spill file open attempts
UNIT	1	Unit number
W	1	Window address (hex)
WAIT	none	Wait for close request
WINDOW	1	Window address (hex)
WRAP	none	Wrap-around command history

ALT Option. The ALT keyword accepts a set of four window dimensions to be used as the alternate window size. The AmigaDOS 2.0 "Zoom" gadget will toggle the window between its current and alternate sizes.

AUTO Option. The AUTO option specifies an auto-open and close window, and implies a close gadget for the window. Instead of opening the window when the console is first opened, the handler waits until a read or write request is received and then opens the window. The window can be closed by hitting the close gadget or CTRL-\ key, except that closing is disabled if a client task has requested the window pointer. The initial activation state of the window can be controlled with the DEACT and INACTIVE options.

BACKDROP Option. The BACKDROP option specifies a backdrop window—one that opens behind all others on the screen.

BPEN Option. The BPEN option accepts a numeric parameter that sets the "block pen" color for rendering in the console window. Under AmigaDOS 2.0 this option has no effect, as windows are rendered using a fixed color scheme.

BRKMODE Option. The BRKMODE option provides flags to control break signals and other internal modes. The control bits currently defined are:

BRKMODE Flag Bit Definitions

Bit	Description
0	Suppresses back-propagation of handler-generated CTRL-C signals
1	Allows uncontrolled access to the window pointer
7	Suppresses all break signals

The flags are specified as a hexadecimal value. For example, BRKMODE#1 suppresses CTRL-C signals at closing, BRKMODE#2 allows any task to request the window pointer, and BRKMODE#80 suppresses all break signals.

CLOSE Option. The CLOSE option requests a close gadget for the console window. The close gadget allows most console operations to be terminated by hitting the gadget.

COLS Option. The COLS option accepts a numeric parameter that sets the default width (in characters) of the display. In most cases the DisplayHandler can automatically detect the size of the console display, but this option allows a specific fixed size to be set.

CUSTOM Option. The CUSTOM option accepts a hexadecimal parameter that specifies the address of the screen to be used. The screen must have been opened by an external application.

DEACT Option. The DEACT option accepts a numeric parameter that specifies the time interval (in seconds) after which an auto-open window will be opened in the inactive state. This is useful to prevent a window from opening at some random time and stealing the activation from the current window. The default interval is four seconds.

DEVICE Option. The DEVICE option provides the device name to be used as the handler's display device. The default is `console.device`, except that CNX: uses `serial.device` by default.

DPEN Option. The DPEN option accepts a numeric parameter that sets the "detail pen" color for rendering in the console window. Under AmigaDOS 2.0 this option has no effect, as windows are rendered using a fixed color scheme.

FONT Option. The FONT keyword allows you to specify a font name for rendering to the console display. The DisplayHandler will attempt to open and use that font for the window. The in-use font is preserved across jumps to another screen.

FONTSIZE Option. The FONTSIZE keyword accepts a numeric font size for the console display. The in-use font is preserved across jumps to another screen.

HLINES Option. The HLines option accepts a numeric parameter to set the size (in lines) of the command history buffer. Setting HLines to 0 will disable the command history.

INACTIVE Option. The **INACTIVE** option specifies that the console window be opened in an inactive state.

LEFT Option. The **LEFT** option specifies that the scrollbar (proportional gadget) be placed in the left-hand window border. The default placement is in the right-hand window border.

LIMIT Option. The **LIMIT** option accepts a numeric parameter and sets the size of the piping buffer. It is relevant only for **PIP:-mode** operations.

KEYMAP Option. The **KEYMAP** option allows you to specify a keymap to use for the console window. Keymaps can be selected on a window-by-window basis, and are preserved across jumps to another screen. The keymap file must have been loaded by the Workbench SetMap program prior to use.

MENU Option. The **MENU** option specifies the name of the menu description to be attached to the console window. The menu description must have been loaded using the **DHOpts** command prior to use.

MINSAVE Option. The **MINSAVE** keyword accepts a numeric parameter to set the minimum line length to be saved in the command history.

NOALT Option. The **NOALT** option suppresses the specification of an alternate size for the window.

NOBORDER Option. The **NOBORDER** option specifies a borderless console window.

NOCLOSE Option. The **NOCLOSE** option suppresses the close gadget on the window.

NODEPTH Option. The **NODEPTH** option suppresses the depth gadget on the console window.

NODRAG Option. The **NODRAG** option suppresses the drag gadget on the console window.

NONBLOCK Option. The **NONBLOCK** option specifies non-blocking type-ahead at the console window. With this option in effect, output to the window proceeds even if there are characters on the command line.

NOPROP Option. The **NOPROP** option suppresses the creation of a proportional gadget (scrollbar) for the console window. Note that it is possible to have a session history without a scrollbar, as the DisplayHandler provides key sequences to scroll through the session.

NOSIZE Option. The **NOSIZE** option suppresses the sizing gadget on the console window.

OVER Option. The **OVER** option sets the default input mode to overstrike.

POPUP Option. The POPUP option instructs the handler to force the display screen to the front whenever it opens the console window.

RAW Option. The RAW option sets the handler to RAW mode. In RAW mode the handler's read stream is not filtered before being passed on to be read, and the stream is not displayed on the console device.

RESAVE Option. The RESAVE option instructs the handler to save unmodified history lines back in the command history buffer.

ROWS Option. The ROWS option accepts a numeric parameter giving the default height (in lines) of the display. In most cases the DisplayHandler can automatically detect the size of the console display, but this option allows a specific fixed size to be set.

S* Option. The S* option specifies the frontmost screen for the display.

SCREEN Option. The SCREEN option supplies the name of the public screen to use for the display. Screen names are case-sensitive. The SCREEN option accepts * as a shorthand for the frontmost screen.

SESSION Option. The SESSION option accepts a numeric parameter to set the size of the session history buffer. Valid sizes range from 2400 to 32000 bytes, with 0 to indicate that no session history is required. The default session buffer size is determined by the total available memory, according to the following schedule:

Default Session Buffer Size (in Bytes)	
Total Memory	Buffer Size
up to 1 Meg	8000
1 to 3 Meg	16000
over 3 Meg	32000

SHARED Option. The SHARED option sets the shared-mode serial flag, and applies only to CNX:-mode operations using the serial.device for the display.

SIMPLE Option. The SIMPLE option requests a simple-refresh console window. This is the default under AmigaDOS 2.0. A simple-refresh window is required to enable the AmigaDOS 2.0 console cut-and-paste facility.

SMART Option. The SMART option requests a smart-refresh console window. This is the default under AmigaDOS 1.3.

SPILL Option. The SPILL option specifies a filename in which to save the console's session history. With the SPILL option in effect, the DisplayHandler writes any excess session history lines to a spill file, and writes the remaining lines in the session to the file when the handler closes. The spill file is opened in the T: directory by default, and the default spill file name is T:Display-Spill.

STICKY Option. The **STICKY** option instructs the handler to retain the current input mode (insert or overwrite) after a line has been entered. By default the handler reverts to its current default input mode.

TRIES Option. The **TRIES** keyword expects a numeric parameter and sets the number of attempts to be made when opening a spill file.

If the **/TRIES** option is supplied, the handler will attempt to create a unique spill file name by appending a number of the form **-NN**, for **NN** from 1 to the **TRIES** count, but will attempt to open the last name in any event. The default number of **TRIES** is 10.

UNIT Option. The **UNIT** option provides a unit number for the display device used by the console handler. Unit numbers are generally required only in conjunction with the **DEVICE** option to request a specific device unit.

W Option. The **W** option is accepted as an abbreviation for the **WINDOW** option as described below.

WAIT Option. The **WAIT** option is used to postpone the closing of the console window until explicitly requested, either by hitting a close gadget or by entering a **CTRL-** character.

This option is typically used to capture the output of a command for temporary use. The **DisplayHandler** scrollbar remains active while the console waits to be closed.

WINDOW Option. The **WINDOW** argument accepts a single parameter providing the (hexadecimal) address for the window to be used with the console. The window must have been opened by another software application, but will be closed by the **DisplayHandler** when the console closes.

WRAP Option. The **WRAP** option selects a wrap-around command history buffer.

Option Examples

The following examples illustrate some of the **DisplayHandler** keyword options. Note that most keywords can be used in combinations with other options.

Examples:

```
CON://640/100/Name/close/auto/bpen#3
CON://640/100/Name/auto/screen**/alt#0,11,640,200/deact#30
CON://640/100/Name/close/sticky/wrap
CON://640/100/Name/auto/screen Phred/hlines 2000
CON:spill/close/font#thin.font           ; use the default size
CON:menu#CON-Menus
CON://640/100/MySession/SESSION#3000    ; a small buffer
CON://640/100/MySession/SESSION#0      ; no session!
```

Old-Style Attributes

For backwards compatibility with the ConMan software used with previous versions of WShell, the DisplayHandler will accept the old-style single-character window attributes recognized by ConMan. If the first option after the window title is not recognized as a keyword, it is assumed to be a set of attributes. However, the recognized attributes set the window options only relative to their normal defaults, rather than toggling the attribute currently in effect. This allows the old-style specification to work in the presence of device-level defaults.

Examples:

```
CON://640/100/Name/cd
CON://640/100/Name/cd/menu#CON-Menus
```

4-5 Console Menus

The DisplayHandler allows user-configurable menus to be attached to the console window. The menu actions can invoke one or more editing or control actions of the handler, or can simply insert text on the command line.

All of the internal editing and control features of the DisplayHandler have a read stream escape-sequence equivalent, and thus can be controlled by a menu selection. Refer to Appendix A for information on these sequences.

The Menu Description File

The DHOpts command will read a menu description file that associates a data string with each menu item or subitem. The data strings can contain both ordinary input (to be placed on the command line) and escape sequences to activate the editing or control features of the handler.

The menu description file is specified by the MENU argument. By default the name given to the menu is the tail name of its description file, but the NAME argument can be used to supply a different name. Menu names are not case-sensitive.

By default DHOpts won't attempt to reload data for an existing menu, but the REPLACE/S switch will force the data to be reloaded. The operation will then fail if the menu is currently in use. If you just want to make sure that a menu is available, use DHOpts without the REPLACE switch.

Menu files are read using a template TYPE/A/N,BAR/S,LABEL,DATA,KEY/K. The TYPE argument specifies the category of the record and uses the value 1 for menus, 2 for items, and 3 for subitems; any other values are ignored. The LABEL argument provides the text for the item label, and is required unless the BAR switch is given. The KEY parameter provides the key shortcut, if desired. The DATA parameter provides the data string; the menu selection acts as a no-op if no string is provided.

The WShell distribution disk includes a sample menu description file. You'll probably find it helpful to study this file carefully before attempting to build a menu file of your own.

Example:

```
1 LABEL "Project"
2 LABEL "New Shell" DATA "newwsh*N"
2 LABEL "Quit" DATA "*E[11]"
```

Attaching a Menu

Once a menu description has been read and processed, the menu can be referenced by a DisplayHandler device name using the MENU keyword. For example, CNC:menu#CNC-Menus would open a CNC: console window with the CNC-Menus menu description attached. Different handlers and windows can share a menu description.

If desired, the preferred menu can be made part of the device-level default when the DHOpts command is run. This default menu can then be overridden by a specific reference to another menu, or removed entirely by omitting the menu name following the MENU option keyword.

Deleting a Menu

The DHOpts command with the DELETE switch will delete a menu description, unless it's currently in use.

Examples:

```
DHOpts MENU S:CNC-Menus NAME std
DHOpts DELETE std
```

4-6 Screen Jumping

The DisplayHandler supports a private packet to allow a window to "jump" to another screen. The jump can be specified as a specific screen (by name with AmigaDOS 2.0) or to the next screen in sequence. WShell 2.0 includes a built-in jump command using this facility; refer to Chapter 6 for more information on jump.

Under AmigaDOS 2.0 the target new screen must be a public screen and is specified by its name. If no name is specified, the "next" public screen in the internally-defined sequence is used. The jump can request that the new window be opened immediately or that it be deferred ("auto-open") until needed.

Windows are automatically resized when jumping to a screen of a different size. The resizing algorithm attempts to preserve the window's aspect ratio with respect to the new screen dimensions.

Examples:

```
1> jump next popup ; jump to next and move it the front
1> jump auto ; stay here, but auto-open
1> jump Workbench ; go to Workbench screen
```

Line Editing and Command History

The ease of use of a command shell depends greatly on its line editing and command history features. A good line editor should make it easy to create and modify commands, and the history facility should let you quickly find and reuse your prior commands. WShell uses the DisplayHandler console handler to provide superb line editing, command history, session history, and a number of other features.

From a systems design standpoint the division of labor between WShell and the input/output console handler is unusual. Command shells on other systems have traditionally been designed with the line editing—if any—built into the shell itself. However, there are a number of advantages to using a special handler, rather than the shell, to implement the line editing and command history facilities. If you run a command that prompts for additional input, you may need to edit these input lines, and the separate shell/handler architecture allows you to do this. It is also useful to maintain a record of these lines, since they are as much a part of your history as the command itself. This effectively reduces the “modality” of the command environment—the number of separate modes of operation presented to the user.

In addition to its superior line editing features, the DisplayHandler console handler was designed to be very flexible and can be extensively customized to suit your personal preferences. Console window attributes, line editing modes, history operation, and most other defaults can be tailored on a device-by-device basis. Chapter 4 describes the options available with the DisplayHandler.

5-1 Line Editing

When you type a character at a console window, the console handler stores the character in a line buffer and displays it in the window. The handler continues to accumulate characters until you press the RETURN key, after which the entered data can be read by WShell or some other program.

As you enter the command line, you can use the many editing functions provided by the DisplayHandler to help build the command. Editing functions are available to position the cursor, delete characters or words, and even insert chunks of text from other commands. These powerful line-editing features make the console very convenient to use.

All of the DisplayHandler editing functions are bound to control keys or function keys, and may be available as menu selections as well. Control keys are entered by holding CTRL (or ALT and CTRL) down while pressing another key. These editing keys would normally be filtered out by the console handler, so they can be used for editing purposes without loss of functionality.

Editing Modes

The DisplayHandler supports two input modes. In *insert* mode, new characters are displayed at the cursor position and everything to the right is shifted over to make room. In *overstrike* mode each new character simply replaces the former one. The console handler assumes insert mode as its initial default, but you can specify overstrike mode as the default by including the `/OVER` option when you open the console window.

CTRL-A—Toggle Input Mode. The CTRL-A key will toggle between the two input modes. That is, if you are in insert mode and you press CTRL-A, you will enter overstrike mode, and conversely.

CTRL-^—Select Insert Mode. The CTRL-^ key will select insert mode unconditionally.

The editing mode normally reverts to its selected default each time you press the RETURN key. For example, if the default is insert mode and you've entered overstrike mode with a CTRL-A, the handler will return to insert mode when you hit RETURN. However, if you include the `/STICKY` option when you open the console, the edit mode will remain set until you change it explicitly using CTRL-A or CTRL-^.

Cursor Positioning

The left- and right-arrow keys can be used to position the cursor anywhere in the command line. You can jump quickly from word to word by using the shifted arrow keys; the shifted left-arrow will position the cursor at the start of the previous word, and the shifted right-arrow will move to the start of the next word.

CTRL-]—Toggle SOL/EOL. The CTRL-] key will toggle the cursor between the start-of-line (SOL) and end-of-line (EOL). It moves the cursor to the SOL if it is initially at the end of the line, and moves it to the EOL otherwise.

ALT-CTRL-I—Skip Forward Name. The ALT-CTRL-I key skips forward to the end of a name, as delimited by a space, /, or : character.

ALT-CTRL-O—Skip Back Name. The ALT-CTRL-O key skips backwards to the beginning of a name, as delimited by a space, /, or : character.

Deleting Characters

Several editing operations are available to delete one or more characters from the active line.

BACKSPACE—Back Up and Delete. The BACKSPACE key moves the cursor to the left and deletes the character there.

DEL—Delete Character. The DEL key deletes the character under the cursor and moves the remaining line to the left.

CTRL-X—Delete Line. As with the standard console handler, the CTRL-X key clears the entire active line.

CTRL-U—Delete to Start-of-Line. The CTRL-U key deletes all of the characters to the left of the cursor, so that the cursor is then positioned at the beginning of the line.

CTRL-Y—Delete to End-of-Line. The CTRL-Y key deletes all of the characters from the cursor position to the end of the line. Note that the CTRL-K key performs a similar action, but saves the characters in the save buffer.

ALT-CTRL-H—Delete Back Word. The ALT-CTRL-H key deletes back to the beginning of a word.

ALT-CTRL-K—Delete Forward Word. The ALT-CTRL-K key deletes forward to the end of a word.

ALT-CTRL-U—Delete Back Name. The ALT-CTRL-U key deletes backwards to the beginning of a name, as delimited by a space, /, or : character.

ALT-CTRL-Y—Delete Forward Name. The ALT-CTRL-Y key deletes forward to the end of a name, as delimited by a space, /, or : character.

Save Buffer Operations

The DisplayHandler has an internal save buffer (sometimes called a “yank” buffer) in which you can store characters. Editing operations are available to copy or cut a line to the save buffer, and to insert the save buffer into the current line. The save buffer persists from line to line until its contents are changed explicitly, so you can use it to perform repetitive operations on multiple lines.

The DisplayHandler provides a “set mark” operation to set a placemaker at the cursor position. Subsequent operations can then act on the “region” defined as the characters between the placemaker and the current cursor position. The mark may be to either the right or left of the cursor in defining the region.

CTRL-@—Set Mark. The CTRL-@ key sets a place marker at the current cursor position. This mark is used to define a region to be copied or cut to the save buffer.

CTRL-K—Cut to EOL. The CTRL-K key sets the mark at the cursor and cuts the remainder of the line to the save (yank) buffer.

CTRL-P—Paste Save Buffer. The CTRL-P key pastes (inserts) the save (yank) buffer at the cursor position.

ALT-CTRL-W—Copy Region to Save Buffer. The ALT-CTRL-W key copies the current region to the save (yank) buffer.

ALT-CTRL-X—Cut Region to Save Buffer. The ALT-CTRL-X copies the current region to the save (yank) buffer and then deletes the region.

Miscellaneous Editing Features

The DisplayHandler offers several other editing and control features that are perhaps less frequently required but still very useful when you need them.

CTRL-T—Transpose Characters. The CTRL-T key will transpose the two characters preceding the cursor.

CTRL-Z—Delete Type-Ahead. CTRL-Z will delete all typed-ahead lines that haven't yet been read by the shell or another program.

CTRL-R—Recall Type-Ahead. CTRL-R provides a similar but less drastic action by recalling the last typed-ahead line and posting it back to the active line buffer.

CTRL-—Undo Line. Sometimes you may be editing a line and accidentally lose it by hitting the up- or down-arrow key. You can then use the CTRL- (control-underbar) key to retrieve the line from the “undo” stack. This restores the former line and even puts the cursor back to its previous position. There is no limit to the depth of the undo stack.

CTRL-\—End-of-File. The CTRL-\ key sends an “end-of-file” indication to the program reading from the console. Some utility programs (such as copy) keep reading until they reach an end-of-file, and you must use CTRL-\ to end the input stream.

Clipboard Operations

The DisplayHandler can read or write characters to the Clipboard, a temporary storage device for passing data between applications. Operations are available to copy all or part of the console's session history to the Clipboard, and to paste the contents of the Clipboard to the command line.

CTRL-V—Paste from Clipboard. The CTRL-V key pastes from the Clipboard to the line buffer.

ALT-CTRL-A—Copy Session to Clipboard. The ALT-CTRL-A key copies the current session history to the Clipboard.

ALT-CTRL-E—Copy Page to Clipboard. The ALT-CTRL-E key copies the current page of the session to the Clipboard.

ALT-CTRL-F—Copy Active Line to Clipboard. The ALT-CTRL-F key copies the active (command) line to the Clipboard.

5-2 Session History

The DisplayHandler saves the text of the lines written to the window as the session history. It provides a scrollbar (proportional gadget) and several keys to enable you to scroll through the session history. Session lines are scanned to remove troublesome escape sequences or control characters, such as form feeds. Harmless sequences, such as those for text or background color changes, are left intact. Note however that the normal action of scrolling the display may split previously paired color changes, resulting in color artifacts in the display.

The session history is used to refresh the window after a resizing event, and the CTRL-W key can be used to refresh the window at any time. The available session control keys are described below.

F3—Move to Bottom of Session. The F3 key moves to the bottom (most recent) page of the session.

F4—Move to Top of Session. The F4 moves the display to the top (oldest) page of the session.

F7—Move Down Session Line. The F7 key moves the display one line towards the bottom (most recent part) of the session.

F8—Move Up Session Line. The F8 key moves the display one line towards the top (oldest part) of the session.

Shift-F7—Move Down Session Page. The Shift-F7 key moves the display one page towards the bottom (most recent part) of the session.

Shift-F8—Move Up Session Page. The Shift-F8 key moves the display one page towards the top (oldest part) of the session.

ALT-CTRL-B—Clear Session History. The ALT-CTRL-B key (or its escape-sequence equivalent <ESC>[66]) will clear the session history.

The session control keys can be bound to other keystrokes or to menu selections for added convenience. For example, ALT-DownArrow and ALT-UpArrow are handy for scrolling through the session history using the arrow keys if bound to the F7 and F8 actions. The FComp filename-completer can be used to remap the keys, and a sample FComp configuration file is provided with key bindings for the arrow keys.

Flow Control

The CTRL-Q and CTRL-S keys are used for flow control. Pressing CTRL-S holds all output to the console, and CTRL-Q releases the output. Since there is no direct visual indication that output to the console has been held, you should be careful not to press CTRL-S accidentally. Output to the console can also be stopped by typing any printing character, unless the handler is operating in the non-blocking mode selected by the /NONBLOCK option.

5-3 Command History

Working sessions with a command shell usually involve the repetitive use of a relatively few commands. Although command shortcuts like an alias list can reduce the amount of typing required to create a command, it is often useful to maintain a “history” of what you’ve done. The WShell command history is automatically enabled when you open the console window using the DisplayHandler. The commands that you enter are saved in the history buffer and can be recalled for later use.

Each console window maintains its own separate history. On a multitasking machine you may find yourself working on several different projects at once, and having a separate history keeps each thread of commands uncluttered with the commands of the others. Those commands that are effectively global in scope can always be defined as a short aliased name.

Retrieving Commands

The history buffer can be visualized as a “stack” with the most recent command on the top and the oldest command on the bottom. Lines can be recalled by using the up- and down-arrow keys. The up-arrow will fetch progressively older commands until it reaches the bottom of the stack. Once you’ve located the desired command, hitting the RETURN key will send it to the WShell. After you press RETURN, the up-arrow key will always retrieve the line you just entered. The shifted up-arrow key will retrieve the earliest history line, and the shifted down-arrow retrieves the most recent line.

Search Keys

While the up- and down-arrow actions are very fast, the DisplayHandler provides an even faster way to locate a line, especially if it’s far back in the history buffer. When you enter a partial command line (possibly just a single character) and then press function key F6, the DisplayHandler searches in the up-arrow direction for lines that match the entered line *up to the cursor position*. Each new line is displayed with the cursor in the same position, so you can press F6 again to search for another match. Function key F5 conducts a similar search in the down-arrow direction.

If you find a line that more nearly approximates the command you’re looking for, you can move the cursor to the right to further constrain the search. After a search the history buffer is positioned at the last line retrieved, just as though you had used the up- or down-arrows to get there.

Clearing the History Buffer

You can clear the command history buffer by pressing the CTRL-B key. This allows you to start with a fresh buffer when you move to a new project and don’t need the existing command history. It could also be helpful in maintaining password security by preventing an unauthorized person from retrieving your prior commands. Clearing the buffer does not release the memory associated with the history buffer.

Command History Options

By default the DisplayHandler will save only those lines that have been modified in the command history. If you retrieve a line and press RETURN without typing any characters, the line will not be reentered into the history buffer. However, the DisplayHandler does support a "true history" mode that will save every line that you enter, regardless of whether it was modified. This operation is selected by specifying the /RESAVE option when you open the console.

Although the history buffer is usually modelled as a stack, you can choose to view it as a circular buffer by selecting the /WRAP option when you open the console. With the WRAP mode enabled, the up- or down-arrow keys will cycle repeatedly through the history buffer. The buffer index "wraps around" to the bottom after hitting the top.

Sizing the History Buffer

By default the DisplayHandler history buffer will hold 50 command lines. You can change this by specifying the desired size (in lines) with the /HLINES option when you open the console. Setting HLINES to 0 effectively disables the command history.

Preloading the Command History

WSH shell includes a utility program called history that can be used to preload the command history, display the current history, or save the history buffer to a file. The argument template for the command is

LOAD/K,NOCLEAR/S,SAVE/K,LIST/S,NUM/S

and there are several modes of operation.

If you enter the history command with no arguments, or with the LIST switch, it will display the current history buffer. For example,

```
R(0); 02:35:14> history
rxxmast
"say hi
rxc
print loadlib.asm
list
validwait
print validwait.asm
df1:
shell
list
run e history.tex
history
R(0); 02:35:15>
```

The NUM option switch will display the number assigned to each history line. DisplayHandler history lines are numbered sequentially from the time the console first opens.

If you specify a filename with the SAVE argument (e.g. `history SAVE ram:capture`), the buffer contents will be written to the specified file. You can then edit this file and keep those commands that you use most frequently.

To add the contents of a file to the current history buffer, specify the history file as the LOAD argument to the history command. For example,

```
R(0); 02:35:20> history LOAD s:SaveHistory
R(0); 02:35:22> history
rexsmast
"say hi
rxc
print loadlib.asm
list
validwait
print validwait.asm
dfi:
R(0); 02:35:23>
```

By default the `history` command will clear the history buffer before loading the new lines, but this action can be suppressed with the `NOCLEAR` option switch. The new lines are always added to the “top” of the history stack, so that they appear as the most recently entered lines.

Swapping History Contexts

You can use the `history` command to simultaneously save the current history and load a new history context; just specify both the `SAVE` and `LOAD` arguments with the command. This can be very useful if you’re running an interactive command, but don’t want to fill up your current history with the lines specific to that command. By saving the current context and loading a new one, you can begin work on a new project with a known set of commands, and then restore your previous environment when you finish.

Example:

```
history save ram:hist load s:texhist
tex
history load ram:hist ; restore prior context
```

5-4 Keymapping

The Amiga’s operating system provides a powerful keymapping facility that can be used to completely reconfigure the keyboard. Keymapping is used primarily to support the various national keyboards, but works equally well as a user-extensible feature. The `DisplayHandler` works with this system keymapping facility, and by means of a suitable keymap you can bind any of the `DisplayHandler` editing functions to the keys of your choice.

What is a Keymap?

A keymap is simply a data table that tells the Amiga's *console device* what characters should be generated with each keystroke. Each time you press or release a key, the keyboard sends a unique *keycode* to the computer. This *input event* filters through several layers of software and may eventually reach your console window. At this point the keymap controls the translation from the keycode to the ASCII characters that your program receives.

Keymaps can become very complicated because of the many qualifier keys that can affect the translation. For example, you can build a keymap that will output different characters depending on whether the SHIFT, CONTROL, ALT, or SHIFT-ALT keys are held down as the key is pressed. Each of these qualifiers effectively multiplies the number of different keys on the keyboard.

Keymaps are stored as files on disk and should reside in your `DEVS:keymaps` directory. To make the keymap available to the system, you must first load it using the `SetMap` program provided with your Workbench disk. This program will load a keymap from disk and make it the default keymap for all currently-open consoles.

One poorly-understood aspect of keymapping is that not all consoles need use the same keymap. You can locally remap your editor's (or WShell's) console to use a different keymap; with the `DisplayHandler` this is done using the `/KEYMAP` option.

Building a Keymap

Since a keymap is just a data table with various pointer references, you can build one in assembly language or any other programming language that allows you to create tables. Volume 1 of the *ROM Kernel Manual* describes the structure of a keymap; it is quite technical but is complete enough to build a working keymap, to which the example keymap `cmap.asm` in the `:sources` directory will attest. You can use `cmap.asm` as the starting point for your own custom keymaps. After editing it to install your key assignments, you must assemble and link it just as you would do for an assembly-language program. Note that although keymaps are structured as executable files, you should never try to run one, as it would likely crash the machine.

Although the direct assembly-language approach has a certain atavistic appeal, most readers will probably want to use a keymap editor tool to build their custom keymaps. Keymap editors know the structure of a keymap and let you manipulate them in a more friendly manner.

Regardless of the means used to construct the keymap, you must first decide which key combinations should be used for which functions. Appendix A lists the editing and control sequences currently implemented by the `DisplayHandler`.

To reassign an editing function to another key, define a keymap that binds one or more editing sequences to the key of your choice. For example, if your keymap emits 9B,35,7E when you press CTRL-up-arrow, then the CTRL-up-arrow key will act just like function key F6. Of course, F6 will continue to act like F6 unless you remap it also. If necessary, you can chain together several sequences to build up the desired one. For example, you can build

a “cursor to end-of-line” key by combining CTRL-], left-arrow, and CTRL-]; in hexadecimal this sequence would be 1D,9B,44,1D.

Loading and Using a Keymap

After you’ve constructed a keymap, whether by using a keymap editor or by directly assembling it, copy it to the `DEVS:keymaps` directory and then load it using the `SetMap` program on the Workbench disk. This will make it the global default keymap; if you don’t want it to be the default, restore the original keymap (also using `SetMap`.) The new keymap can then be specified with the `/KEYMAP` option in the `DisplayHandler` console definition.

Input Event Handlers

In addition to the console-based keymapping facility, the Amiga also provides a way to define *input event handlers* that can act as global keymappers. This is frequently used for “hot-key” programs that respond to your command regardless of which window is activated. By means of suitable software you can bind text strings to various key combinations and create keystring macros that work across several applications.

The `FComp` filename-completer supplied with `WShell` uses an input event handler to trap keystrokes, and it can be used for keymapping as well as filename completion. The `FComp` configuration file allows you to assign a string to a keycode and qualifier combination, and this string is sent to the console handler whenever the key is pressed. The keymap string can contain just a plain command line, or it can use the editing sequences in Appendix A to create new editing features or to reassign the editing keys.

Although an input event handler “sees” all keystrokes in the system, `FComp` was designed to trap keystroke events only when a `WShell` window is active, so it won’t interfere with other event handlers that you may wish to use. Refer to Chapter 12 for more information on using `FComp`.

Chapter 6

Resident and Built-In Commands

It's often advantageous to have frequently-used commands preloaded in memory, both to avoid unnecessary disk searches and to provide faster execution. WShell provides a *resident list* for this purpose, which supports both executable commands loaded from disk as well as the predefined *built-in* commands implemented as part of WShell.

There are several advantages to making a command resident rather than just copying it to a "memory" or RAM disk. Loaded commands take up somewhat less memory space than the equivalent disk file, and will begin executing more quickly, since they don't have to be relocated. In addition, the resident list is checked before searching for macro programs or disk-based executable files, so this search time is eliminated. However, resident commands do not survive rebooting the computer, so you may want to keep some commands on a recoverable RAM disk as well.

The resident list is searched automatically whenever you enter a command at a WShell. The search for a resident command follows the check for the previously-loaded module, but precedes the search for ARexx macro programs and other executables. The resident list is therefore a very fast and convenient method for loading commands which are used frequently and for which no macro "front end" is required.

The search for resident commands can be bypassed by preceding the command with the "search-escape" character [. For example, the command `[delete df1:badboy` will load the `delete` command from disk without checking the resident list. Refer to the Chapter 2 for more information on the search-escape character.

Under AmigaDOS 2.0 the WShell resident list actually consists of two separate lists, one private to WShell and one maintained as part of AmigaDOS. The two lists are complementary in several respects. The WShell list supports "load-on-call" commands and provides a more robust checksumming capability, and as a private list is visible only to other WShells. The AmigaDOS list is potentially accessible to any application in the system and can be modified under program control via DOS calls. In general the dual resident list search will be transparent to the user.

The WShell private resident list is searched first, so any command placed on this list will override a command of the same name in the AmigaDOS resident list.

The AmigaDOS resident list is managed by the `resident` command included as part of AmigaDOS. Refer to your AmigaDOS 2.0 software documentation for information on the usage and capabilities of the `resident` command.

6-1 Code Purity Requirements

There is an important restriction on the commands that can be used as resident modules. WShell allows only resident modules implemented as “pure” code, which means that the program does not modify its loaded binary image as it executes. WShell checksums each module when it is first loaded and after every execution to make sure that the program code has not been corrupted.

Unfortunately, many existing programs do not follow the simple requirements necessary to ensure code purity. For example, most “C” programs will flunk this checksum test unless they have been specifically designed to be “residentable.” If a resident command does fail the checksum test, WShell will report the wayward program with a message that reads

Bad checksum for resident module *program-name*.

The simplest way to determine whether a program is residentable is to make it resident and then run it a couple of times. If the warning message doesn’t appear, it’s probably safe. Most of the AmigaDOS 2.0 commands in the C: directory are safe as resident commands, but with prior versions of AmigaDOS the `list`, `delete`, and `copy` commands were not safely residentable.

The AmigaDOS resident list does not currently provide a checksum, though it may in the future. However, WShell will still compare the checksums of a module loaded from the AmigaDOS list before and after execution, and will report any differences with the warning message

*****Impure resident module.**

Should you receive such a warning, it’s advisable to remove the offending command from the resident list.

6-2 Maintaining the Resident List

The WShell resident list is maintained using the built-in `resi` command, which can install, delete, display, or reorder the resident modules. The argument template for `resi` is

,,,,,,,AS/K,-AUTO/S/B,-LIST/S/B,-DELETE/S/B,-IGNORE/S/B

The keywords with the /B (“brief”) modifier can be abbreviated as desired; for example, `-AUTO` could be given as just `-A`.

Only executable programs may be specified as targets for the command, and up to nine programs can be specified at once. The command name can include an explicit path, as in `resi c:run`, and the path is stripped off before the command is placed in the list. If you don’t specify any arguments, the `resi` command will display the currently-loaded resident commands.

The initial set of resident commands can be defined using the WShell configuration file, which is processed when the first WShell opens. Refer to Chapter 11 for information on using the configuration file.

Whenever you modify or display the resident list the `resi` command sorts the resident list so that those commands used most frequently are near the top. The modules are sorted by their reference (usage) count. Each WShell records its accesses to the resident modules

by incrementing the usage and in-use counts, so that the usage count is a running total of the number of accesses to each module. The usage count is limited to a maximum of 127.

Resident modules are automatically replaced when you specifying an existing name with the **resi** command. For example, if **list** is already resident, **resi c:list** will replace it. However, the built-in commands must be explicitly deleted before they can be replaced by another command. This is to prevent a built-in command from being replaced accidentally, as there is then no way to restore the original command.

The AmigaDOS resident list is maintained separately from the WShell list using the **resident** command. Refer to your AmigaDOS manual for more information on **resident**.

Auto-Loading Resident Modules

By default the **resi** command loads each resident program when the command is run, even though you may not need to use the resident commands immediately. However, if you specify the **-AUTO** switch, the command name is added to the resident list, but the program code is not loaded until you first use the command. These "load-on-call" resident commands take less time to define and also save memory, as only those commands actually used are loaded.

Alternate Names for Resident Commands

If you wish to give a resident command a name other than its filename, you can specify the **AS** keyword followed by the desired name. For example, **resi c:status as st** would make **status** resident with the name **st**. Only a single command name can be given if the **AS** keyword is used.

Deleting Resident Commands

A resident command can be deleted by specifying the **-DELETE** option of the **resi** command, as in **resi -d avail**. Provided that the command is not in use at the time, it is removed from the resident list and the storage space is returned to the system.

Built-in commands can be deleted as well, but once such a command has been removed, there is no way to restore it until WShell is reconfigured.

Ignoring Checksum Failures

If you receive the dreaded checksum failure message for a resident command, the only safe course of action is to delete the program from the resident list and run it only from disk. However, if you have been assured that the program is actually safe to run as a resident command, or are simply inclined towards experimentation, you can try replacing the command with the **-ignore** option in effect. This will suppress the checksum comparison, but may also lead to mysterious crashes.

Listing the Resident Commands

You can display the resident commands by specifying the `-list` option, or by simply issuing the command with no arguments. The `-list` option will display a table similar to this:

Resident Module	Usages	In-Use	Status
CD	26	0	BUILTIN
copy	20	0	LOADED
delete	18	0	LOADED
NewWSH	13	0	LOADED
Alias	9	0	BUILTIN
History	6	0	LOADED
makedir	6	0	LOADED
Jump	6	0	BUILTIN
version	3	0	LOADED
status	3	0	LOADED
protect	3	0	LOADED
Recall	2	0	BUILTIN
Resi	2	1	BUILTIN
type	1	0	LOADED
ExecIO	0	0	AUTO
RunWSH	0	0	AUTO
which	0	0	AUTO
SwapCD	0	0	BUILTIN
Stdin	0	0	BUILTIN
Skip	0	0	BUILTIN
REXX	0	0	BUILTIN
PushCD	0	0	BUILTIN
PopCD	0	0	BUILTIN
Mounted	0	0	BUILTIN

The **Usages** column shows the total number of times you've used the command. The **In Use** count shows the number of command shells that are currently running the command and is usually 0. In the above example only the `resi` command was in use at the time. The **Status** column indicates whether the command is built-in, auto-loaded, or already loaded into memory.

You can specify one or more (partial) names with the `-list` option, in which case the display will show only those commands that match the supplied names. In the above example `resi -list c r` would display

Resident Module	Usages	In-Use	Status
CD	26	0	BUILTIN
copy	20	0	LOADED
Recall	2	0	BUILTIN
Resi	2	1	BUILTIN
RunWSH	0	0	AUTO
REXX	0	0	BUILTIN

Note that all of the "c" entries are displayed before any of the "r" lines.

6-3 More on Code Purity

For the technically inclined, there are several classes of programs that will fail the purity test but may still be useful as resident commands. Those programs that merely write constant values into their global data structures (such as base addresses for permanently-loaded libraries) will be unconditionally safe as resident modules. They can be run repeatedly and simultaneously from several shells with no improper results.

Programs that write values that are constant within the program, but that vary from one shell to the next, will work correctly as long as only one shell at a time runs the code. The great majority of "C" programs fall into this class, as the standard startup code used by the "C" compilers writes values such as the input and output streams and stack pointer into the global data structure. These values will be corrupted if two shells attempt to run the program simultaneously, generally with confusing or disastrous results.

A small number of programs use a trick called "seglst splitting" to support special requirements like background operation or to bundle several executables into a single file. Needless to say, since part or most of the program seglst disappears, the checksum for the seglst does not remain invariant. These programs generally don't need to be (or shouldn't be) run more than once, and so are not good candidates for resident status anyway.

For example, the `rexxmast` executable used to launch the ARexx server uses seglst-splitting to create a background program. Only one instance of the `rexxmast` program can be active at a time, since the program manages a unique message port and must therefore verify that the server is not already active.

6-4 The Built-In Commands

WShell includes a number of predefined or "built-in" commands. Instead of being loaded from a disk file when required, these commands are part of the WShell program and are always available. The built-in commands are placed on the WShell resident list when WShell is configured, and can be displayed with the `resi` command.

The selection criteria for the built-in commands were based on the assumed frequency of use, the nature of the command itself, and the likelihood that a custom user-supplied command would be preferred. Many of the built-in commands are used to modify values internal to the shell, such as `alias`, `failat` or `prompt`, or are used as part of the batch command facility, such as `if` or `quit`. Other commands, such as `jump`, require access to private information maintained by the shell. By coding the commands very tightly and allowing them to call on functions already used in WShell, more than two dozen commands were implemented in about 4,000 bytes of code. This is substantially less than the same commands would take as separate programs in your C: directory.

Most users should find the built-in commands to be very convenient in their operation. However, if a particular command doesn't meet your needs, you can delete it from the resident list using the `resi -delete` command.

The remainder of this section is devoted to individual descriptions of the built-in commands. In most cases the commands are closely modeled on their AmigaDOS counterparts, and will function in the usual manner.

Note that although all of the built-in commands are described here, the ones actually available on the resident list will depend on the version of AmigaDOS installed on your system. AmigaDOS 2.0 provides a number of built-in commands of its own, and whenever the AmigaDOS and WShell commands were virtually identical in operation, the WShell command has been withheld in favor of the system-provided command.

Most of the built-in commands expect one or more arguments on the command line, for which WShell uses a command-line prompting and parsing facility similar to the one used by AmigaDOS. The command template is displayed in response to entering the command with just a ? argument, after which you can supply the requested arguments.

The usage templates given below are the informal requirements for the commands. When prompting for arguments the commands display the actual template as a series of keywords followed by option characters. For example, the formal template displayed by the `alias ?` command is

```
NAME,=,LITERAL/K/F,LOCAL/S,-LIST/S/B,TRUNC/K,-KILL/S/B:
```

Commands provide a *return code* when they exit, which is generally interpreted as an error severity level. A command should return a value of 0 if it completed successfully, and most of the built-in commands use a return code of 20 to indicate failure. In certain cases a lesser number (e.g. 5) will be used to indicate a warning condition.

ALIAS

Usage: Alias [NAME] name [= | LITERAL] value [GLOBAL] [LOCAL] [NOECHO] [-LIST] [-KILL] [TRUNC]

The `alias` command is used to maintain the alias list, and is described in Chapter 7.

Return Codes:

20 if invalid arguments were specified

Examples:

```
R(0);Workbench:> alias CLone "copy □ clone"
```

CD

Usage: CD [directory]

The `CD` command changes the shell's current directory to the specified directory, or displays the current directory name if no argument is supplied. The previous current directory will be stacked if the `AUTOPUSH` configuration option is in effect.

Under AmigaDOS 2.0 the directory name may be supplied as a wildcard pattern. If the literal directory name doesn't exist, the wildcard pattern is expanded to check for a unique directory name. An error message is issued if the two or more directories match the wildcard pattern. Refer to your AmigaDOS 2.0 documentation for information on wildcard patterns.

Return Codes:

20 if an invalid directory was specified

20 if an ambiguous wildcard name was specified

See Also: `POPCD`, `PUSHCD`, `SWAPCD`

Examples:

```
R(0);Workbench:> cd df1:
R(0);ShellDev2:> cd
ShellDev2:
R(0);ShellDev2:>
```

ECHO

Usage: ECHO *string* [NOLINE] [FIRST] [LEN]

By default the echo command displays the argument string on the standard output stream with a "newline" (line-feed) appended. The argument must be enclosed in double-quotes if it includes blanks or "white space." The automatic newline can be suppressed by specifying the NOLINE switch.

The FIRST and LEN arguments allow you to display a substring of the argument. FIRST gives the index of the first character to be displayed, and should be between one and the length of the string. Zero or negative values of FIRST are set to one, and values greater than the string length are set to the length.

The LEN argument gives the number of characters to display, exclusive of the newline, and should be non-negative. Zero or negative values of LEN are set to the full string length. If LEN is specified without FIRST, the last LEN characters of the string are displayed; this gives you a way to access a right substring without knowing the string length in advance.

The echo command is generally used only within script files, but it can be issued interactively. When used with output redirection it is a simple and convenient way to write a line to a file. For example, the environment variable ENV:path could be redefined with the command echo >env:path df1:c,c:.

Return Codes:

20 if invalid arguments were specified

Examples:

```
R(0);ShellDev2:> echo "This is a test"
This is a test
R(0);ShellDev2:> echo "Hi *NBill"
Hi
Bill
R(0);ShellDev2:> echo "abcde" FIRST 2
bcde
R(0);ShellDev2:> echo "abcde" FIRST 2 LEN 2
bc
R(0);ShellDev2:> echo "abcde" LEN 2
de
```

ELSE

Usage: ELSE

The **else** command begins a group of commands to be executed as the alternative to the preceding **if** command. That is, the **else** commands are executed only if the **if** test was false. The range of the **else** command extends to the next matching **endif** command.

Return Codes:

20 if the matching `endif` was not found

See Also: `IF`, `ENDIF`

Example:

```
if warn
    echo "Warning error"
else
    main df0:testdata
endif
```

ENDCLI

Usage: `ENDCLI`

This command can be used to terminate a WShell session. The `endcli` command is not necessary if you've attached a close gadget to the window, as hitting the close gadget is a more convenient way to close the shell. You can also close the WShell by hitting the `CTRL-\` key one or more times; the required count is set by the `EOFLIM` configuration option.

Be careful when using the `endcli` command (or close gadget) that you don't inadvertently close your last command shell. Unless you've loaded Workbench or have an "instant CLI" background program, you'll then have to reboot the computer.

Example:

```
R(0); 02:44:19> endcli
```

ENDIF

Usage: `ENDIF`

The `endif` command terminates the range of the nearest preceding `if` command. It is useful only within a script (batch) file, but does not generate an error message if it is entered interactively.

See Also: `IF`, `ELSE`

Example:

```
if error
    echo "Bad news"
endif
```

ENDSKIP

Usage: `ENDSKIP`

The `endskip` command terminates the range of a preceding `skip` command. It is useful only within a script file.

Return Codes:

5 if a skip was terminated

See Also: `SKIP`

Example:

```
skip
echo "You won't see this"
endskip
```

FAILAT

Usage: **FAILAT** [*rc*]

The **failat** command is used to set the level at which a command is considered to have failed. The nominal failure levels used within AmigaDOS are shown in the table below.

Return Code	Severity Level
5	Warning
10	Error
20	Failure

These error levels are useful in categorizing errors by their apparent severity. Whenever the return code from a command in a script file exceeds the current failure level, the command stream is abandoned and a message is issued. By modifying the failure level within the command file, you can retain control after an error that would otherwise abort the sequence. This is often useful to allow files to be deleted or a more informative error to be issued.

The default failure level is 10, and this value is restored at the end of each command sequence. The **failat** command can be issued interactively, but is not very useful except within script files.

Return Codes

20 if an invalid return code was specified

See Also: **QUIT**

Example:

```
failat 25
cc main -o obj/main.o
```

IF

Usage: **IF** [*NOT*] [*WARN* | *ERROR* | *FAIL*] [*EXISTS filename*] [*[VAL] str cond str*]

The **if** command is used to conditionally execute a series of commands. The command arguments are examined for one or more test conditions, and if any of the test conditions are true, the command lines following the **if** up through the next **else** or **endif** command are executed. The **NOT** keyword can be used to reverse the result of any of the test conditions.

- **WARN**
Tests whether the return code from the last command matches or exceeds the warning level of 5.
- **ERROR**
Tests whether the previous return code was ≥ 10 .
- **FAIL**
Tests whether the previous return code was ≥ 20 .
- **EXISTS *filename***
Tests whether the specified file exists.
- ***str cond str***
The condition *cond* must be one of EQ, GE, or GT. The command compares the two strings to check whether the specified condition is satisfied. By default a string comparison is

performed, but the VAL switch will force a numeric comparison. The null string can be specified as "".

The arguments for a comparison test are normally used as literal values, but if the argument begins with a \$ character, it is considered as the name of an environment variable. The command substitutes ENV: for the \$ and then reads the operand value from the environment variable.

The if command is valid only within a command file, and an error message will be issued if it is entered interactively.

Return Codes:

- 20 if invalid arguments were specified
- 20 if used outside of a command file
- 20 if the test was false but no endif was found

See Also: ELSE, ENDIF

Example:

```
if <file> EQ ""
    echo "No filename specified"
    quit 20
endif
```

JUMP

Usage: JUMP [SCREEN screen-name] [NEXT] [POPUP] [AUTO] [DELAY time]

The jump command allows you to move or “jump” the shell’s window to another screen, while preserving the window’s command and session histories. The target screen can be specified by its public screen name (with AmigaDOS 2.0 only) or as the next one in sequence. If no screen is specified, the window moves to the Workbench screen.

The popup switch requests that the target screen be brought to the front before opening the shell window there. The auto switch specifies that the shell window should open only when required for reading or writing.

The delay argument specifies a time delay (in system ticks) before the command returns to the shell.

The auto switch in conjunction with a delay argument is useful for temporarily jumping a shell window off of a screen. For example, when changing Workbench screen modes under AmigaDOS 2.0, you must remove all foreign windows from the Workbench screen to allow the screen to be reconfigured. The jump auto delay 150 command works nicely for this purpose.

Screen jumping is actually implemented within the WShell DisplayHandler, and is therefore available only when the shell window is managed by the DisplayHandler.

Return Codes:

- 20 if an invalid argument was specified
- 20 if an invalid console handler is in use
- 20 if the jump failed

Example:

```
jump screen PHred
jump next auto delay 120
```

LAB

Usage: LAB [*name*]

The lab command is used to label a specific line within a command file, which can then be used as the target of a skip command. This command is useful only within a script file, but no error message is issued if it is entered interactively.

See Also: SKIP

Example:

```
if warn
    skip end
endif
main test
lab end
```

MOUNTED

Usage: MOUNTED *filename*

This command can be used to check whether a given file, device, or ASSIGNED name is currently mounted. It is intended primarily for use within script files or REXX macro programs. The device status is checked “quietly”—without posting a requestor demanding that the user mount (or ASSIGN) the device. The return code from the command is zero if the device is mounted, and 5 (warning level) if the device is not mounted. Note that mounted doesn’t check whether a file exists, but only whether the device has a volume mounted.

Return Codes:

- 5 if the device is not mounted
- 20 if the arguments were invalid

Example:

```
mounted df1:
if warn
    ask "Please mount disk in drive 1"
endif
```

POPCD

Usage: POPCD

This command restores a previously “pushed” directory and makes it the current directory. It issues a warning message if no directory is available to be “popped.” If the AUTOPUSH configuration option is in effect, a popcd command can be used after any directory change to return to the previous directory.

Return Codes:

- 5 if there were no stacked directories

See Also: CD, PUSHCD, SWAPCD

Examples:

```
R(0);Workbench:> pushcd df1:
R(0);ShellDev2:> popcd
R(0);Workbench:> popcd
Nothing to "pop"
R(20);Workbench:>
```

PROMPT

Usage: PROMPT prompt-string

This command defines a string of characters to be used as your “prompt”—the indication that the WShell is awaiting your next command. In the simplest case the prompt can be just a 1> reminder like the standard CLI, but WShell provides greatly enhanced prompting capabilities. The prompt can consist of plain text and special keywords, identified by a preceding % character, that are substituted with their current value at the time that the prompt is displayed. For example, the %t keyword indicates the time-of-day, so entering the command prompt "It's %t> " might prompt with It's 02:38:14> .

Because of the expanded functionality of the WShell prompts, a separate chapter has been devoted to the subject. Refer to Chapter 8 for more information.

Return Codes:

20 if an invalid argument was specified

Example:

```
R(0);01:12:33> prompt "R(%r); %c> "
R(0);Workbench:>
```

PUSHCD

Usage: PUSHCD [directory]

The pushcd command saves the current directory on a special internal stack and optionally selects a new current directory. If no directory argument is specified, the current directory is pushed on the stack and remains in effect. If a directory is specified, it becomes the new current directory. There is no limit to the number of directories that can be stacked. Each pushcd can be undone with a corresponding popcd command.

This command has no AmigaDOS counterpart. It is very useful when you need to temporarily visit a far-away directory and plan to return to your current directory thereafter. Note that WShell also provides an AUTOPUSH option that effectively does a pushcd whenever you change directories. Refer to Chapter 11 for information on using AUTOPUSH.

Return Codes:

20 if an invalid directory was specified

20 if a memory allocation failure occurred

See Also: CD, POPCD, SWAPCD

Example:

```
R(0);Workbench:> pushcd df1:
R(0);ShellDev2:> popcd
R(0);Workbench:>
```

QUIT

Usage: QUIT [*rc*]

The **quit** command causes an immediate exit from a script file with the specified return code. The optional argument (which must be a non-negative integer) allows you to specify the return code for the exit; the default return code is 0. This command is generally used only in script files, but can also be used interactively. Using the **quit** command from within a nested script file will cause an unconditional exit from all levels of the script.

Return Codes:

20 if an invalid return code is specified

Example:

```
if <file> EQ ""
    echo "no filename given"
    quit 25
endif
```

RECALL

Usage: RECALL *line-number*

The **recall** command is used to retrieve a command from the history buffer by its assigned history number. After displaying the next prompt, the shell recalls the specified line and then posts it to the command line. The line numbers for each command in the command history can be displayed using the **history num** command.

The line-recall facility is available only when the shell's console window is being managed by the DisplayHandler.

Return Codes:

20 if invalid arguments were specified

Example:

```
ShellDev2:> history num
1. cd
2. avail
ShellDev2:> recall 1.
ShellDev2:> cd
```

RESI

Usage: RESI *filename* [*,filename*] [*AS name*] [*-AUTO*] [*-LIST*] [*-DELETE*] [*-IGNORE*]

The **resi** command maintains the WShell resident list, and is described in an earlier section of this chapter.

Return Codes:

20 if invalid arguments were specified

20 if the command couldn't be loaded

Example:

```
1> resi c:status -auto
```


REXX

Usage: REXX command-line

The **rexx** command is used to force a command to be executed as a REXX macro. REXX-language macros are executed transparently as part of the search sequence for a command. However, WShell will not permit a REXX macro to execute itself recursively, in order to prevent accidental recursion loops. Since controlled recursion can be very useful, the **rexx** command is provided to unconditionally execute a REXX program.

The **rexx** command can be used only if the ARexx resident process is active. Refer to Chapter 10 for more information on using ARexx with WShell.

Examples:

```
R(0);01:22:58> rexx fact 5
R(120);01:22:59> rexx fact 6
R(720);01:23:00>
```

SKIP

Usage: SKIP label [/BACK]

The **skip** command is used to advance the position in a script file. It is valid only within a script file, and an error message will be issued if it is entered interactively. The argument must specify a label within the script file, and the command assumes a forward reference unless the BACK switch is given. If the label can't be found, the shell will skip to the end of the script file and issue an error.

Return Codes:

20 if used outside of a script file

20 if the label can't be found

See Also: LAB

Example:

```
if warn
  skip end
endif
main test
lab end
```

STACK

Usage: STACK [stack-size]

The **stack** command is used to set or display the stack size (in bytes) for the shell. If the size argument is given, it is installed as the new stack size; otherwise, the command displays the current stack size.

"Stack" in this context refers to a preallocated memory area given to each program when it is first launched. WShell acquires a block of memory for the stack before it runs a command, and then releases the stack when the program finishes. Most Amiga software is designed to run with only the 4,000 byte default stack, but some programs expect much larger amounts—sometimes 20,000 bytes or more in extreme cases. Failure to specify a sufficiently large stack usually causes an abnormal termination, and in some cases will crash

the machine. Check the documentation for your software applications for the recommended stack size.

The **stack** command will not let you set an unreasonably small stack size. If you specify less than 2,000 bytes of stack, the command issues an error message and leaves the stack size unchanged.

If the **CHECKICON** configuration option has been specified, WShell will use the stack size in a program's .info (icon) file instead of the default stack. Refer to Chapter 11 for more information.

Return Codes:

20 if an invalid argument was specified

Examples:

```
R(0); 03:12:03> stack 10000
R(0); 03:12:05> stack
Current stack size is 10000 bytes
R(0); 03:12:06>
```

STDIN

Usage: STDIN

The **stdin** command reads from its standard input stream and runs the lines as commands in the current shell, just as though they had been executed from a batch file. Unlike a batch file, no parameter substitutions or other changes are made to the commands.

The primary use for the **stdin** command is to allow a command stream to be piped to an existing shell, instead of having to open a new shell.

Examples:

```
1> echo "status" | stdin
```

SWAPCD

Usage: SWAPCD

This command interchanges the current directory and the topmost stacked directory, thus providing a convenient way to "toggle" between two current directories.

You must have at least one stacked directory (from a previous **pushcd** command) before you can use **swapcd**. Alternatively, if the **AUTOPUSH** option is in effect every directory change will add the old directory to the stack. The **swapcd** command issues a warning message if there are no stacked directories.

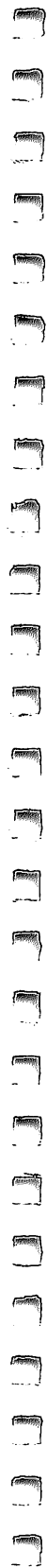
Return Codes:

5 if there were no stacked directories

See Also: **CD**, **POPCD**, **PUSHCD**

Examples:

```
R(0);Workbench:> pushcd df1:
R(0);ShellDev2:> swapcd
R(0);Workbench:> swapcd
R(0);ShellDev2:>
```



Chapter 7

Command Aliases

Have you ever wished for a shorthand notation for commands that are frequently used, easy to misspell, or just hard to remember? WShell provides such a facility in the *alias list*, a flexible and convenient way to define command shortcuts.

The alias list is a series of entries each consisting of a name and a corresponding value. Every time you enter a command, WShell searches the alias list for an alias name matching the command name (the first “word” on the command line.) If it finds a match, the corresponding alias value is substituted in place of the command name. The name matching is not case-sensitive, so an alias `abcd` would match the command `AbcD`. The alias value can contain any characters that would be valid on a command line, including command options and redirection specifiers. Normally only one level of alias substitution is performed, but a local alias may be substituted again from the global alias list.

The alias substitution is the first step in processing the command. After a substitution has been performed, the command line is processed as it would be normally. For example, suppose that the alias name `dm` had the value `resi -d`. If you entered `dm avail`, then the command actually executed would be `resi -d avail`.

7-1 Maintaining the Alias List

The WShell alias list must be maintained using the `alias` Built-In command. The `alias` command has several modes of operation, as it can be used to define new names, update existing values, delete names, or display the list. The full usage template for the `alias` command is

`NAME[, =, LITERAL/K/F, GLOBAL/S, LOCAL/S, NOECHO/S, -LIST/S/B, -KILL/S, TRUNC/K`
and the most common forms of the command are:

1. `alias name = value`

This form creates a new alias `name` (if it doesn't already exist) and assigns the value string to it. The `=` sign is optional, and the value string must be enclosed in double-quotes if it includes spaces or special characters. Alternatively, the value can be preceded by the `LITERAL` keyword, in which case the entire remainder of the command line is taken as the value.

2. `alias name`

This form of the command will delete (remove) a name from the alias list.

3. `alias`

Executing the command with no arguments will display the current contents of the list in the form `name = value`.

4. `alias -KILL`

The `-KILL` keyword will delete all entries from the list, so use it with caution.

Alias names are usually defined as part of the WShell configuration file, but can be added or modified at any time. There is no fixed limit to the number of aliases that you can define, although it is unlikely that you will need more than a few dozen.

7-2 Local and Global Aliases

WShell supports both local and global aliases to provide you with maximum flexibility in defining your commands. As the terms imply, local aliases are visible only within the shell in which they were defined, whereas global aliases are immediately visible in all shells. By default an alias is local in scope, except that those defined in the WShell configuration file are always global. The `alias` command provides a `GLOBAL` switch to override the default and a `LOCAL` switch to force local scope.

Local aliases are inherited whenever a new WShell is created by `newwsh`, `runwsh`, or by a piped command. Local aliases are resolved first and so take precedence over the global definitions, and a local alias may refer to a global alias. In the example below the command `one` is resolved to `two` in the local list and to `three` in the global list:

```
3> alias one = two           ; a local definition
3> alias GLOBAL two = three ; a global definition
3> one                       ; I enter 'one' and get 'three'
--> three
```

A global alias can be specified by including the `GLOBAL` keyword in the command, as in the following example:

```
3> alias GLOBAL one = two
```

7-3 Argument Substitution

When an alias substitution is made, normally the entire alias value is inserted verbatim in place of the command word. However, if the value string contains the character sequence `[]`, the arguments to the command are inserted in place of the `[]` pair. For example, in the lines below the arguments `df0:c RAM:` have been inserted in place of the `[]` characters in the alias.

```
3> alias clone = "copy [] clone"
3> clone df0:c RAM:
--> copy df0:c RAM: clone
```

If the alias is being substituted into a compound command line, the command arguments are defined as that part of the command up to the next piping or background specifier character.

The `[]` argument substitution convention is easy to use and is adequate for many requirements, but on occasion you may need to define an alias that provides for multiple

substitutions. This can be done easily using an inline ARexx program, as in the following example:

```
3> alias myp = LITERAL "parse arg $1 $2 $3;'mypgm' in $1 out $2 opts $3"
```

The statement `parse arg $1 $2 $3` splits the command argument into the three variables \$1, \$2, and \$3, and these variables are then used to construct the desired command. Refer to Chapter 10 for more information on using ARexx with WShell.

7-4 Echoing Alias Commands

If you've set the `ENV:echo` variable (or a local `echo` variable) to `ON`, WShell will display the resolved command after each alias substitution (if any) has been performed. For example,

```
3> dm avail
--> resi -d avail
3> alias link "blink main.o+subs.o library :lib/amiga.lib to main"
--> alias link "blink main.o+subs.o library :lib/amiga.lib to main"
3> link
--> blink main.o+subs.o library :lib/amiga.lib to main
3>
```

You can limit the echo display to just those lines that were modified by alias substitution with the configuration option `CHANGED`; refer to Chapter 11 for more information. The automatic echoing of an alias substitution can be suppressed by specifying the `NOECHO` switch with the `alias` command.

7-5 Listing Options

The `alias` command with no arguments, or with the `-LIST` keyword, will display the alias list. By default the both the local and global definitions are shown, with a leading (L) or (G) to indicate the type of entry. You can use the `GLOBAL` and `LOCAL` switches to limit the scope of the display.

The output lines are normally formatted to fit nicely on a standard width console, assuming that your alias names are reasonably short. Any values that are too long to fit will be truncated and an ellipsis (...) appended. You can specify the truncation limit with the `TRUNC` keyword, and setting the value to 0 or less will disable truncation completely. The truncation length can also be supplied in the `ENV:trunc` environment variable or `trunc` local variable.

```
3> alias -L TRUNC 30
3> alias -L TRUNC 0 >dh0:saveall ; capture everything
```

7-6 Cyclic Definitions

Although WShell performs only one level of alias substitution, it is possible to define compound statements that will result in endless cycles of alias substitution. For example, defining

```
3> alias try1 = "runwsh try2"
3> alias try2 = "runwsh try1"
```

and then entering `try1` will result in a locked cycle of alias resolutions. You should avoid such definitions. Incidentally, it is possible to break out of the above cycle without rebooting the computer. Can you figure out how to do it?

7-7 Command Abbreviations

The alias list can also be used to specify the allowable abbreviations for each command. Command abbreviations could of course be defined by simply listing all of the desired equivalences, but this brute-force approach is not very elegant, and the alias list would then become long and unwieldy. WShell instead uses the alphabetic case of the alias name to define which substrings of the name are permissible as abbreviations. The rules for defining the abbreviations are very simple:

1. The command name must match a leading substring of the alias.
2. All leading uppercase characters of the alias must be matched.
3. The character following the matched part must be in lowercase.

For example, suppose that you wanted some abbreviations for the `makedir` command, but still wanted the `make` command to remain as `make`. If the alias `MAKeDir = makedir` were in effect, then the shell would accept `ma`, `mak`, `maked`, and `makedi` as abbreviations for `makedir`, but it would not recognize `m` or `make`. The presence of the uppercase `D` in `MAKeDir` prevents `make` from being accepted, by application of rule #3.

In the example above, `mk` and `mkd` would not be accepted as valid abbreviations, by application of rule #1. If you want to use abbreviations that are not leading substrings of a command, these must be defined as separate aliases. For example, the alias `MKDir = makedir` would further allow `mk`, `mkd`, `mkdi`, and `mkdir` to be used as aliases for `makedir`.

The alias entry for an abbreviated command is processed in exactly the same way as any other alias match, so such aliases can include command-line arguments in the value string as well. For the purposes of abbreviation processing, all non-alphabetic characters are considered to be in uppercase. WShell always uses the first match that it finds in the alias list, so you should be careful to define unambiguous abbreviations.

Chapter 8

The Prompt String

As you would expect of a command shell, WShell issues a prompt string to notify you that it's ready for your next command. In some command shells this prompt is limited to just a dull 1> or such, but WShell allows you to display useful information as part of the prompt. Of course, you can define the string to be anything you want it to be, so if you prefer having the shell just say *Howdy!*>, it will do so.

8-1 Prompt Keywords

The prompt string can consist of ASCII text, ANSI escape sequences, and WShell keywords. Keywords are identified by a % followed by one or two alphabetic characters, and each keyword has an associated value field. Each time WShell is ready to display the prompt, the prompt string is interpreted, and any keywords are replaced by their associated values. The keywords that are currently defined are:

Table 8.1 Prompt String Keywords

Keyword	Substituted Value
%B	Vertical bar
%C	Current Directory (c=full, C=short name)
%D	Date (AmigaDOS 2.0 only)
%E	Elapsed Time
%F	Failure Level
%G	Greater Than >
%H	Home Directory
%I	Task ID
%L	Less Than <
%MC	Chip Memory (c=bytes, C=K bytes)
%MF	Fast Memory (f=bytes, F=K bytes)
%MP	Public Memory (p=bytes, P=K bytes)
%N	Task Number
%R	Return Code
%S	Stack Size
%T	Time of Day (t=military, T=civil)
%W	Original Window Title
%Y	Error Code ("Why?")
%1-%7	Select Colors 1-7

The prompt string can consist of text and the above keywords in any order, and can include standard ANSI escape sequences for text positioning or setting the display mode. The total length of the prompt string itself must be less than 60 characters, and the expanded string (after all substitutions are made) must be less than 120 characters. Except where noted, the prompt keywords are not case-sensitive.

Prompt strings can be installed using the built-in `prompt` command or the AmigaDOS equivalent. The entire prompt string must be enclosed in double-quotes if it contains blanks

or other “white space.” To include a “newline” in the prompt, you can use the `*N` escape combination.

Typical uses for the prompt include displaying return codes, timing programs, or showing the current directory.

Examples:

```
1> prompt "R(%r); %c%g "  
R(0); ShellDev:> prompt "%t %e> "  
02:25:17 2.00>
```

8-2 Prompt Programs

No fixed list of prompt options could hope to satisfy all Amiga users, so WShell supports an even more general form of prompting. The prompt string can include the name of a program enclosed in square brackets, as in `%[isrexxx]`. Each time the shell scans the prompt, the specified program is executed just as though it had been entered from the command line. If your program requires command-line arguments, these too can be enclosed in the brackets. A prompt program can be either an executable or an ARexx macro program, and can of course be written in any language that you could use for programs run from the command line (but note the advice section below.) The first test of this feature was rather humorous: the TxE~~d~~ editor (named E) got caught in the prompt, so it got fired up after every command!

Not everyone will need this feature, but here are a few of the things you could do with a prompt program:

- check whether ARexx is currently active, or
- ring a bell, using the Amiga’s audio synthesizer, or
- check on the status of a background task.

Since whatever program you place in the prompt string will probably be run more often than all other commands put together, try to keep it small and fast. If possible, you should put the prompt program on the resident list, as you’ll find it annoying to have a slow program in the prompt. Prompt programs should not return an error code, because the shell will abandon the remaining prompt if an error occurs. Also, note that only one program can be placed in the prompt, so if you have ten little side-effects that need to be updated at every prompt, put them all into a single program.

8-3 Variable and Backticked Command Expansion

In addition to the special prompt keywords and commands recognized by WShell, you can also include references to local or environment variables and backticked commands in your prompt string. These are expanded before the prompt is displayed, in the same way as they would be in the command line itself. Refer to Chapter 2 for information on variable expansion and backticked commands.

For example, a prompt string of “\$rc ‘date’> ” would be expanded to the return code from the last command (\$rc) and the current date (‘date’). Note that backticked commands are run in a subshell rather than the current shell, and so may behave differently than the same command run directly in the prompt.

8-4 Titlebar Prompts

Although the prompt string can present you with all sorts of useful information, there may not be enough room on the console to display it all, particularly if you intend to type a command after the prompt. One solution to this problem is to use the window’s titlebar as an additional message area. Although it’s easy to write a program to display the current directory or other information in the window title, WShell gives you a more convenient way to do this.

If you create an environment variable ENV:titlebar (or a local titlebar variable) and place a prompt-like string in it, the resulting (expanded) string will be automatically displayed in the window’s titlebar area. Any of the prompt string variables can be included in this titlebar string, with the exception of the command field %[. The window title is updated after each command has been run, so the information displayed there will be as recent as the last command execution. Simple titlebar strings can be created by just writing to the environment variable. For example, the command

```
echo >ENV:titlebar "Process %n with stack %s >>>%c"
```

will display the shell task number, your available stack, and the current directory in the window title.

The window title is not updated unless it actually changes, in order to minimize the “flicker” that results when a new title is posted to a window. Placing variables that change frequently (like the time of day) will of course require more frequent updates, so you may wish to use only those fields that change relatively infrequently. The total length of the expanded titlebar string is limited to 120 characters.

The rendering for the titlebar is a little different from that in the console window itself. In particular, ANSI escape sequences are not interpreted in the window title, although such sequences won’t actually cause any harm. Since the window title is rendered using only two colors, the color selection keywords (which are translated to ANSI sequences) will have no effect.

Remote Titlebar Updates

Updating the window’s titlebar normally requires that the shell have access to the actual window pointer, a situation not possible if the shell is being run over a network with the display on another machine. If WShell is unable to obtain a window pointer from its console handler, it will issue an AmigaDOS ACTION_SET_COMMENT packet specifying the new titlebar string. This allows the titlebar to be set over the network, provided that the local display supports this packet (as the DisplayHandler does.)



Chapter 9

Command Piping

On occasion you may wish to use the output of one command as input to another command. For example, you might want to generate a list of files using the `list` command and then display each file using the `type` command. This could be done by redirecting the output of the first command into an intermediate file and then using this as the input stream, but WShell provides a more convenient mechanism called *piping*. As the name implies, a pipe can be thought of as a two-ended data stream. Information can be written into one end of the pipe and then read from the other end. The reading and writing can take place concurrently, so that the reader can access each line as soon as it has been written.

Piping is specified on a command line by placing a vertical bar “|” between two or more commands entered on a single line. Whenever the piping character is recognized, WShell opens a special *pipe handler* and obtains an input and output stream from it. It then creates a new background shell task and passes the current command to the background shell, and then continues processing the remainder of the original command line. The background task uses one end of the pipe as its output stream, and the other end of the pipe is used as the input stream for the remaining command. For example, the command

```
writepgm -s myfile | readpgm
```

would run the `writepgm` command with the specified arguments and feed its output to the standard input for the `readpgm` command. WShell runs the background piping tasks at the same priority as the shell that spawned them.

An important point to note here is that both commands will run concurrently. If the reader program gets ahead of the writer program and empties the pipe, it will simply wait for more data to come. Conversely, if the writer gets too far ahead of the reader, the pipe fills up and the writer's task will be blocked (suspended) until the reader catches up. Eventually the writer program will terminate and close down its end of the pipe, and shortly thereafter the reader program will finish and close its end of the pipe.

Note that the standard AmigaDOS commands define the vertical bar as part of their wildcard pattern specification. In such usage the | character will almost never have a space preceding it, so WShell uses the presence of a space to distinguish between these two cases. If you need to change the piping character, you can specify an alternate character with the `PIPING` option in the WShell configuration file. Refer to Chapter 11 for more information.

9-1 The PIP: Handler

Since piping must look like ordinary input or output to the reader and writer programs, respectively, the piping must be processed by an AmigaDOS handler. WShell was designed to work with either the `PIP:` device implemented as part of the `DisplayHandler` or with the more generic `PIPE:` handler supplied with AmigaDOS. WShell first tests whether the `DisplayHandler PIP:` device is available, and if not attempts to open the `PIPE:` device. It assumes that `PIPE:` refers to a named pipe and generates a unique filename for each instance of piping.

Although the two pipe handlers are similar in their operation, there are a few subtle differences. The PIP: handler treats each block of data as a distinct record, and will return a short block to a reader task if the current record is shorter than the read request. It also supports back-propagation of CTRL-C break signals when a writer task writes to a “broken” pipe—one that no longer has a reader. This allows a system of multiple piped commands to be terminated with a CTRL-C signal instead of always running to completion.

To use the PIP: device, you’ll need to define PIP: using the DHOpts command, as for example with DHOpts PIP:. If you prefer to use the AmigaDOS PIPE: handler, make sure that PIPE: has been mounted (following your AmigaDOS documentation) and don’t define the PIP: device. The standard startup sequence for AmigaDOS 2.0 mounts PIPE: by default.

9-2 Designing Programs for Piping

Not every command can be used sensibly in a piping system. Programs that are designed for such uses are frequently called “filters,” as they transform an input stream into an output stream, possibly adding or removing information in the process. There are three basic design requirements for such filter programs:

1. The reader program should read from its standard input stream until an end-of-file is reached,
2. The writer program should write to its standard output stream, and
3. The writer program should exit if it receives a CTRL-C signal.

The last condition is important to allow the pipe to close down prematurely. If the writer continues to write to the pipe after the reader has closed its end, the pipe handler will deliver a CTRL-C break signal to the writer’s task. Of course, no great harm will occur if the writer ignores this break signal; it simply wastes time.

Once these basic requirements have been satisfied, the filter program can perform almost any sort of data transformation.

9-3 Example Filter Programs

Since AmigaDOS does not provide piping support with its Shell or CLI, it’s not surprising that few of the standard AmigaDOS commands are useful as piping filters. The WShell software therefore includes some examples of filter programs in the :filters directory of the distribution disk. We’ll look at a couple of these here.

Some of the filters are written as executables and others are REXX-language macro programs. Although macro programs are not as fast, there can be a distinct advantages to using REXX filter programs. In particular, you can then use the tracing facilities to help debug your programs—even single-stepping through them if necessary. Chapter 10 has more information on using REXX-language macros.

In addition to the programs in the :filters directory, the STDIN built-in command is intended primarily as a piped command. It reads a series of commands from the standard

input stream and executes the commands within the current shell, much as though they had been read from a batch file. Refer to Chapter 6 for information on the STDIN command.

A Pipe “Tee”

Suppose that you want to save the output of a program for later use, but that you also want to see it on the console. This can be accomplished by piping the output of the command to the **Tee** filter, and then redirecting the output of **Tee** into a file. The command line would look like this:

```
first | Tee >savefile
```

The **Tee** filter simply reads each line from the standard input and then writes it to both the console and the standard output stream.

Head

Sometimes you may only need to see part of a program's output, such as the first few lines of a directory listing. The **head** filter reads and displays ten lines from standard input and then exits. For example, `list quick | head` will show you the first ten files in the current directory.

9-4 The ExecIO Utility

The WShell package includes **ExecIO**, a multi-purpose filter program whose basic operation is a line-oriented copy between an input and output stream. It provides options that make it useful for selecting subsets of the input data or for searching for certain character strings. **ExecIO** is not limited to writing just to a file; if invoked from within an **ARexx** macro program it can transfer information directly into the macro program's variables.

ExecIO is useful for searching and selecting records apart from its operation as a piping filter program, but is described here because of its many uses as a filter.

Command Arguments

The **ExecIO** command follows the AmigaDOS argument conventions using the template `READ/K,WRITE/K,FROM/K,FOR/K,FIFO/S,LIFO/S,STEM/K,VAR/K,LOCATE/K,AVOID/K, COLSTART/K,COLEND/K,QUOTE/S` whose keywords are described below.

The **READ** and **WRITE** arguments specify the input and output files, respectively. The default input is the standard input stream, and the default output is the standard output stream.

The **FROM** argument must be numeric and specifies the first line number to be transferred to the output. The default value is one (the first line.)

The **FOR** argument is also numeric and specifies the total number of lines to be written to the output. **ExecIO** returns a warning error of 6 if a **FOR** count was supplied but not enough lines were available to satisfy it.

The **FIFO** option specifies that the command output is to be queued to the current console handler. The **LIFO** option is similar and specifies that the output is to be stacked (last-in, first-out order) in the console stream.

The **STEM** and **VAR** arguments are applicable only if the command was issued from within an **ARexx** macro program, and only one of the arguments can be used. **STEM** supplies a stem variable—for example, **OUT.**—to receive the output lines, and the lines are installed in the consecutive elements **OUT.1**, **OUT.2**, and so on. The final line count is placed in **OUT.0**. The **VAR** option provides a variable name to receive a single line of output. An error will result if multiple output lines are passed to the **VAR** argument.

The **LOCATE** argument specifies a search string to be used to qualify the input lines. The search is case-insensitive and accepts only those input lines containing the specified string. The similar **AVOID** argument specifies a search string used to reject input lines.

The **COLSTART** and **COLEND** arguments must be numeric and specify a starting and ending column for the search process. The default values are one and the line length, respectively.

The **QUOTE** switch specifies that AmigaDOS quoting should be applied to each output line, such that the line could be read back in as a command argument. For example, the line **My File Name** would become **"My File Name"** after applying the required double-quotes. The quoting algorithm is smart enough to recognize and escape any special characters in the line. **QUOTE** allows **ExecIO** to be used as a filter for lines that must be subsequently interpreted as AmigaDOS commands.

Most combinations of arguments are permitted with **ExecIO**. The **FROM** and **FOR** arguments can be used with any other arguments, but only one of **WRITE**, **FIFO**, **LIFO**, **STEM** or **VAR** can be specified. Only one of **LOCATE** and **AVOID** can be specified. **COLSTART** and **COLEND** can be used with other arguments, but have no effect unless **LOCATE** or **AVOID** was specified.

Note that the **FIFO** and **LIFO** options are valid only if the output stream supports the **ACTION_QUEUE** and **ACTION_STACK** DOS packets. The **DisplayHandler** console handler included in the **WShell** package meets this requirement, as does the AmigaDOS 2.0 **CON:** handler.

ExecIO Examples

The following examples illustrate the uses of the **ExecIO** command. Refer also to the program **IOTest.rexx** in the **:rexx** directory for more usage examples.

Line-oriented Copying. The command **execio from 12 for 20** will copy from standard input to standard output, beginning with line 12, for a total of 20 lines.

Queueing Commands. A file of commands can be queued for processing by **execio read commfile fifo**. The commands can be stacked in last-in, first-out order using the **LIFO** option in place of **FIFO**.

STEM Argument. The STEM argument places the “output” of the command into a compound variable in a macro program’s symbol table, and is valid only if ExecIO was invoked from an ARexx macro program. For example, `list devs: | execio stem out.` would pipe the output of the `list` command into the variables `OUT.1`, `OUT.2`, and so on, with the line count in `OUT.0`.

VAR Argument. The usage of the VAR argument is similar to that for STEM, and is valid only if ExecIO was invoked from an ARexx macro program. A single line of output is placed in the specified variable. For example, `cd | execio var currentdir` would place the output of the `cd` command in the variable `CURRENTDIR`.

Selecting Lines. LOCATE (or AVOID) can be used to select certain lines from a file or stream. For example,

```
ExecIO read waitforport.asm LOCATE move
```

would display all lines containing `move` in the file `waitforport.asm`.

Searching. Using LOCATE with FOR 1 provides a search command that returns a success or failure indication. The return code will be 0 if at least one line was found, or 6 if no lines matched the LOCATE string.



Chapter 10

Using REXX-Language Macros

Most command shells provide some sort of “scripting” language so that you can write command programs instead of entering complex or repetitive commands. WShell supports two such script languages—the standard AmigaDOS “execute” scripts, and a much more powerful macro-language facility using ARexx.

ARexx is an implementation of REXX, a high-level language that was specifically designed for macro-processing applications. REXX has many features that you probably wouldn’t expect to find in a macro language, including full structured control statements (DO i=1 to 10; DO WHILE ...;), compound IF statements, an extended case-selection construct SELECT; WHEN ...; WHEN ...; OTHERWISE ..., full arithmetic capabilities, an extensive function library, and many more. REXX programs are highly readable and will be immediately familiar to anyone who has used a block-structured language such as C, PL/I, or Pascal.

REXX macro programs are executed transparently by WShell, so the user does not need not know whether a given command is a macro program or an executable file. Command scripts written in REXX consist of language statements processed by an interpreter and *host commands* that are executed by the shell. Macro processing is relatively fast, since the language interpreter is resident in memory.

Since WShell supports ARexx as its “native” language, programs written in REXX can be used just as though they were executable programs. This allows macro programs to be written as “front ends” for other programs (possibly macros themselves) in order to transform the input or output into the required form. For example, suppose that a certain command had an awkward or unfamiliar syntax that made it difficult to use. By writing a macro to invoke it, you could define your own command syntax or switch options and transform them internally to the required command. Similarly, if the output of a command is not correctly presented, a macro could capture the output stream and reformat it.

WShell automatically detects the presence of the ARexx server by looking for the well-known message port “REXX”. While the ARexx server is active, each command line is passed to the server and analyzed to see whether it’s a REXX macro program. The shell program then waits at its subcommand message port for commands issued from the macro program, and processes them as they are received. Eventually the invocation message returns carrying a return code and error number, and the shell returns to its normal command mode.

In this chapter we’ll look at some of the ways that ARexx can be used with WShell. A full tutorial on the language is beyond the scope of this manual, but several references are provided at the end for those who wish to learn more about REXX.

10-1 Activating the ARexx Server

The ARexx server process must be active before REXX macros can be used with WShell. WShell automatically detects the presence of the server by looking for a public message port called "REXX"; if this port does not exist, the shell skips to the next step in its search sequence.

You can activate the ARexx server by issuing the `rexmast` command supplied with the ARexx distribution disk. This command should be placed in your `startup-sequence` file if you'll be using ARexx frequently. Once started, the server remains installed in the background until you reboot the machine. However, if you need to deactivate the ARexx server, just issue the `rxs` command. This will prevent any further REXX macro invocations, although the server won't actually terminate until the last REXX program finishes.

10-2 Invoking REXX-language Macro Programs

Once you've activated the ARexx server, WShell automatically looks for a REXX program corresponding to the commands you enter. The search for a REXX program is part of the standard search sequence for WShell, and occurs immediately after checking for a resident command but before checking for an executable program file. This allows you to write REXX macros that "front" for an executable file.

Note that support for REXX macros is transparent—you don't have to precede the program name with a command to identify it as a REXX program. However, if you want a REXX program to call itself (a *recursive call*), you must use the `rexx` built-in command to launch the program. This safeguard is in place to prevent an accidental recursion loop, since REXX programs are frequently given the same name as the executable program that they invoke.

There are a few other cases in which you may need to use the `rexx` command to invoke an ARexx program. WShell examines each command line before passing it to ARexx in order to screen out certain special cases. Command names ending in a colon `:` or slash `/` are almost always directory names intended as an implicit `cd`, so these commands are not automatically forwarded to ARexx. In addition, commands enclosed in double-quotes, but not containing a semicolon, are assumed to be quoted filenames rather than inline ARexx programs and therefore are not forwarded to ARexx.

The `:rexx` directory of the distribution disk has some sample REXX programs in it. Even if you did not purchase ARexx with WShell, you may want to take a look at these programs to get a feel for the language.

10-3 The REXX Search Path

The search path for REXX programs is different from that used by WShell. When you send a command to ARexx, it begins searching in your current directory and then proceeds to the system REXX: device.

The file name assumed for the REXX program depends on how you specified the command. If you include an explicit file extension, as in `dir.rexx`, only that name will be used in the search. Otherwise, the search looks for both the command name with the

extension `rexx` and for the unextended name. For example, the command `dir shell` would cause a search for the files `dir.rexx` and `dir` in the current directory and in `REXX:`.

Similarly, the search path depends on whether or not you specify an explicit device and directory in the command name. The command `df1:rexx/dir testdir` would look only in the directory `df1:rexx`, whereas `dir testdir` would look in the current directory and in the `REXX:` directory (which might be the same as `df1:rexx`, of course).

Early Search Termination

There is one case in which ARexx will terminate the search for a file without checking the entire search path. If it finds a non-REXX file with the same name as the search target, ARexx assumes that this is the desired file and terminates the search. To extend the above example, if a file named `dir` exists in the current directory, and it isn't a REXX program, the search will terminate before checking in `REXX:`.

10-4 Capturing Output from Commands

ARexx macro programs frequently need to capture the output from an AmigaDOS command for further processing. Although there are several ways to do this, it's often most convenient to use WShell's piping in conjunction with the ExecIO program available in the `:filters` directory. ExecIO has a direct interface to a macro program's variables and can copy from its input stream into a stem variable that you provide; refer to Chapter 9 for more information on ExecIO.

In the following example we wish to reprocess the output of the AmigaDOS `path` command to show the extended WShell search path. The `execio` command is used twice, once to capture the output of the `path` command, and again to read the `ENV:path` variable. In both cases the output lines are placed in a stem variable and then displayed in a loop.

```

/* Displays the WShell extended path      */
call pragma('W','Null') /* no requestors! */

if exists('ENV:path') then
  do
    'path | execio stem L.'
    'execio read ENV:path stem G.'
    say L.1
    if L.0 > 2 then
      do
        say 'Local:'
        do i=2 for L.0-2 /* skip lines */
          say '  'L.i
        end
      end
      say 'Global:'
      do i=1 for G.0
        say '  'G.i
      end
    end
  end
else 'path' /* No ENV:path ... */

```

10-5 Inline REXX Programs

Occasionally you may need to write an *ad hoc* macro program for which it hardly seems worthwhile to invoke your editor, create the program, save it to disk, and then run it. ARexx supports a convenient quoting convention that allows you to express a program as a "string file" by simply enclosing the entire program in double-quotes. Rather than looking for a file containing the program, the ARexx interpreter uses the command line itself as an inline program. An inline program can even include arguments and input/output redirection.

Although this facility is most often used for only very brief programs, you can write some surprisingly powerful programs with just a few REXX statements. For example, the inline program

```
"wc=0;do until eof(stdin);wc=wc+words(readln(stdin));end;say wc" <infile
```

will count the number of words in the file infile.

The closing double-quote is not actually required to indicate an inline program unless there are arguments being passed to the program. In certain cases it may be advantageous to omit the closing double-quote. Since WShell classifies commands enclosed in double-quotes but not including a semicolon as probable filenames, rather than as inline programs, such programs would normally require the use of a preceding *rexx* command. Omitting the enclosing double-quote makes the intended use as an inline program clear. For example, the commands

```

1> rexx "say 12**3"
1> "say 12**3
1> "say 12**3;"

```

will all run the inline program say 12**3. In the first example the `rexx` built-in command was required to force the interpretation as an inline program. The second and third examples are automatically passed to `ARexx` because of the omitted closing quote and included semicolon, respectively.

Inline Aliases

Using inline REXX programs becomes even more convenient when coupled with the WShell alias facility. The alias list lets you predefine your commonly-used inline programs and then refer to them with a single command. Any number of such programs can be defined, and since the alias value string can be quite long, you're not limited to programs that fit on a single command line. For example, the command

```
alias nh LITERAL "'history |execio stem L.';do i=1 to L.0; say i L.i;end"
```

defines an alias `nh` that displays the command history with line numbers. It uses the alias `LITERAL` keyword so that double-quotes can be embedded in the value string.

If the alias list makes `ARexx` more convenient to use, then `ARexx` returns the favor by making aliases much more powerful. For example, although argument substitution in aliases is limited to a single string, you can use a REXX program to allow multiple argument substitution. In the following alias definition

```
alias fmt LITERAL "parse arg $1 $2;'format DRIVE' $1 'NAME' $2"
```

the two arguments are assigned to the variables `$1` and `$2` and then embedded in the `format` command.

Here's another brief example that uses an inline REXX program to define a multiple-command alias. The alias command

```
alias wp LITERAL "pushcd DH0:wp;wp &;popcd"
```

first uses `pushcd` to save the current directory and move to `DH0:wp`, then runs the `wp` command in the background, and finally restores the current directory using `popcd`. Remember that commands issued from an `ARexx` program are processed like any other WShell command and so can refer to the alias list.

"Resident" REXX Programs

We can extend the preceding ideas further by considering an inline `ARexx` program as a sort of "resident" REXX program. Now suppose that we have an `ARexx` program that normally resides on disk, but we want to make it resident on the first usage. This can be done easily by writing the program so that it first defines an alias to do all the work, and then invokes the alias. In the example that follows, called `popall.rexx`, the program issues an alias command to define `POPA11` and then invokes it.

```
/* Pops all pushed locks. Defines an alias to do the work, */  
/* and then invokes it. WSH, 10/89 */
```

```
'alias POPA11 '"do until rc > 0;''popcd >NIL:'';end*"''  
'popall'
```

10-6 Communicating with Other ARexx Hosts

Simple macros usually need to issue commands only to the shell that actually invoked the macro program, and so don't need to know anything about other ARexx hosts in the system. However, on occasion you may want to write macro programs that communicate between two or more host applications. This is the key to using ARexx as a software integration tool; in principle, any two programs that can talk to ARexx can arrange to talk to each other.

ARexx uses public message ports to support its command interface. Any program that supports the ARexx interface will open a public message port called its *host address*. For example, each WShell opens a port named WSH_ with the task number appended to it, so that shell task number 3 would open a message port named WSH_3.

The first prerequisite to communicating with another ARexx host is to learn its host address—the name of its public message port. If the host is a “server” process that can accept commands from anyone at any time, this can be a fixed “well-known” port name. The ARexx server is an example of such a process; only one such server is ever active, and its port is named “REXX”.

In contrast, some hosts are expected to provide a private environment so that a series of commands can be executed. WShell falls into this category. It opens the host message port as part of its initialization process, but reads from it only when a macro program is active. The macro program is thus granted exclusive access to the shell's environment for the duration of its execution. If the macro program transfers access rights to an external host by passing its host address, then that external host can use the shell's resources. This handshake protocol can be summarized in the following example:

1. An external host invokes a WShell to run a handshake macro program as its initial command, and passes its own host address as an argument to the macro.
2. The macro program acknowledges the external host by sending it a command message including its own host address (e.g. WSH_5), and then waits for the command to be replied.
3. The external host now uses the shell's resources through the host address message port, and replies the command message when it finishes.
4. The macro program issues an `endcli` to the shell and then exits.

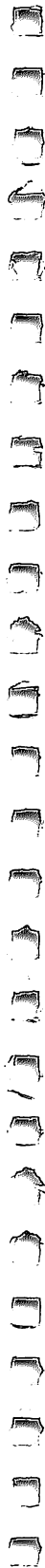
10-7 Learning More About REXX

If you purchased ARexx with WShell, you should have received a copy of the *ARexx User's Reference Manual* with it. This manual covers all of the standard REXX language facilities as well as the Amiga-specific extensions made in ARexx, but is intended as a reference rather than a tutorial.

The language standard is set forth in *The REXX Language: A Practical Approach to Programming* (Prentice-Hall, 1985) by M. F. Cowlishaw. This book is highly recommended, as it offers both a clear exposition of the language as well as an interesting glimpse of the factors that influenced the language design.

A more tutorial approach to the language is available in *Modern Programming Using REXX* (Prentice-Hall, 1985) by R. P. O'Hara and D. R. Gomberg. The program examples assume that you're running them on an IBM mainframe computer (the only implementation available at the time), but most of the programs should work with ARexx as well.

A recent book on ARexx providing both tutorial and reference material is available from Abacus Books of Grand Rapids, MI. *Using ARexx on the Amiga* by Chris Zamara and Nick Sullivan includes many programming examples to help with learning REXX, as well as a major section on controlling external applications from within an ARexx program. You should be able to order a copy from your Amiga dealer.



The Configuration File

The WShell software was designed to accommodate the needs of a diverse group of users and thus has many configurable options. Some of these options can be changed at any time—for example, by means of an environment variable setting—but others are best set once and then left alone. This latter group are controlled by the WShell *Configuration File*, a text file that describes the desired option settings.

The WShell configuration file is called `S:Config-WShell` and is read when the first WShell opens. This ensures that a known environment is established whenever WShell is first activated, regardless of whether this occurs when you boot the system or at a later time.

The configuration facility supplants the `SetWSH` command distributed with previous versions of the WShell software, and `SetWSH` should no longer be used.

Configuration Records

Each line in the configuration file is considered as a single record, and the records are identified by an initial keyword. Four record keywords are recognized: `options`, `special`, `alias`, and `resi`, and the keywords are not case-sensitive. Although the records are structured as a command line to the shell, they are processed just as simple text strings. The records may appear in any order.

For improved readability the configuration file can also include blank lines and comments, identified by an initial semicolon (;). Trailing comments are permitted as well, as the following example

```
options autopush changed ; display changed lines
```

illustrates.

WShell performs some error checking on the configuration records and reports any errors with a message of the form `Error in file S:Config-WShell line 12`. Such errors are considered to be non-fatal, but generally cause the offending line to be ignored.

Modifying the Configuration

The configuration file can be modified at any time, but the changes will not take effect until the next time the WShell library is opened. If you've modified the file and want WShell to read it, you must first close down all instances of WShell, including any background tasks, and then flush `wshell.library` from memory. This can be done by issuing the command `stack 12000000` from a plain CLI, by using the "flushlibs" menu option from Workbench, or of course by rebooting the computer. If you now reopen a WShell console, the new configuration options should be in effect.

11-1 The OPTIONS Record

The OPTIONS record supplies the cache limit, EOF limit, and several true/false options. The argument template is

`CACHELIM/K,EOFLIM/K,AUTOPUSH/S,CHECKICON/S,NOBATCH/S,CHANGED/S,ONEBACK/S,
PATHNAME/S,SPECARG/S,LATECD/S,NOIMPCD/S,SHORRTCD/S,NOBREAK/S`

and only a single OPTIONS record should be present.

CACHELIM. This numeric argument sets the maximum program size that can be automatically cached between commands. Commands to be cached must still meet the requirement for “residentability” in addition to meeting the CACHELIM size limit. The default value is 10,000 (bytes), and setting the limit to zero will disable command caching.

EOFLIM. This numeric argument sets the number of consecutive ends-of-file (EOFs) required to close a shell. An EOF is signalled by entering CTRL-\ at the console window. The default value is five, and setting EOFLIM to one will emulate the (non-configurable) action of the AmigaDOS 2.0 Shell.

AUTOPUSH. This switch option enables the automatic pushcd command for every change in the current directory, whether made by the built-in cd command or an implicit cd.

CHECKICON. This switch tells WShell to launch a program using the stack size from the program’s .info (icon) file. WShell updates the stack size only if the specified amount is larger than the value currently in effect.

NOBATCH. This switch is used to suppress the echo display of commands read from a batch file, and is applicable only if the Echo local or environment variable is on.

CHANGED. This switch is used to limit the echo display to those commands resolved in the alias list or otherwise transformed. It is applicable only if the Echo local or environment variable is on.

ONEBACK. This switch tells WShell to ignore the background character (by default an ampersand &) except at the end of the command line.

PATHNAME. This switch tells WShell to provide a full pathname for the executing command, which allows the command to locate the directory from which it was loaded. For example, if the command `format` was loaded from `df0:system`, WShell would provide `df0:system/format` as the command name.

SPECARG. This switch allows the shell special characters `<`, `>`, `|`, `&`, `[`, `'`, and `;` to be used as command arguments if preceded by the escape character. In addition, the escape character itself (usually `*`) can be escaped. Normally such characters can be escaped only within double-quotes, but for passing commands to external systems (for example, over a network) the double-quotes may not be acceptable.

LATECD. Implicit CD commands are normally recognized before searching the shell's current and path directories, thus giving a directory name precedence over an executable command of the same name. Setting the LATECD option will postpone the test for implicit CD commands until after the (mounted) path directories have been searched. This emulates the (non-configurable) action of the AmigaDOS 2.0 Shell.

NOIMPCD. For reasons beyond the author's ken, some users do not like the implicit CD feature introduced by WShell. The NOIMPCD switch is for them: it completely disables the test for an implicit directory. *De gustibus non disputandum.*

SHORTCD. A command shell must maintain the full name of its current directory as part of its CLI data structure. Normally this name is an absolute path, and in a deeply nested directory the absolute path may exceed the space allotted for the directory name. The SHORTCD option tells WShell to build the directory name as a "short path", a path name terminating (if possible) at an assigned name rather than at the root (volume) name. For example, if WORKING: were assigned to MyVolume:Projects, then the short path for MyVolume:Projects/Source would become WORKING:Source.

NOBREAK. When you enter a break signal (control-C,D,E, or F) at a DisplayHandler console, the signal is normally sent to the last active client task for the handler. This allows a background shell—launched by run or the & character—to receive break signals if it's actively writing to or reading from the console. The NOBREAK option prevents background shells from receiving break signals from the DisplayHandler.

Example:

```
options cachelim 20000 autopush changed
```

11-2 The SPECIAL Record

This record establishes the special characters used by WShell. The argument template is
ESCAPE/K,SEARCH/K,PIPING/K,RUNBACK/K
and only one SPECIAL record should be present.

ESCAPE. This keyword lets you change the character used as the "escape" inside a double-quoted string. The default character is the asterisk (*) used by AmigaDOS. Note that the new escape character goes into effect immediately.

SEARCH. This keyword specifies the "search-escape" character, which is used to suppress the search for resident, built-in, and ARexx macro commands. The default search-escape is the left-bracket ([) character.

PIPING. This keyword specifies the character used to indicate command piping. Note that the piping character is recognized as such only when it is preceded by a space. The default character is the vertical-bar (|).

RUNBACK. This keyword specifies the character used to indicate a background command. When placed at the end of a command, the background character indicates that the command is to be run instead of being executed directly. Background characters, like piping

characters, are recognized as such only when preceded by a space. Multiple background characters will be recognized in a command line unless the **ONEBACK** option is in effect. The default character is the ampersand (&).

Example:

```
special escape * search [ piping | runback &
```

11-3 The ALIAS Record

The **ALIAS** record is essentially the same as the built-in **alias** command, and thus uses the same argument template

NAME,=, **LITERAL/K/F**, **GLOBAL/S**, **LOCAL/S**, **NOECHO/S**, **-LIST/S/B**, **TRUNC/K**, **-KILL/S/B** described in Chapter 7. Note however that the **LOCAL**, **-LIST**, **TRUNC**, and **KILL** keywords are not relevant in the configuration context and therefore should not be used. Aliases defined in the configuration file are always global.

Any number of **ALIAS** records may appear in the configuration file, and you'll probably find this to be the most convenient way to initialize the alias list. The **alias** command can still be used to modify the alias list, of course.

Examples:

```
alias MAKEDir = mkdir
alias tron "echo >ENV:echo ON"
```

11-4 The RESI record

The **RESI** record is very similar to the built-in **resi** command, and uses the same argument template

```
,,,,,,,,,,AS/K,-AUTO/S/B,-LIST/S/B,-DELETE/S/B,-IGNORE/S/B
```

described in Chapter 6. As in the case of the **ALIAS** record, some of the keywords are not relevant in the configuration file and are therefore not accepted. In this case the **-LIST** and **-DELETE** keywords should not be used, and the **-AUTO** switch is implicit.

Any number of **RESI** records may appear in the configuration file, and this is the recommended way to initialize the resident list.

Example:

```
resi c:path c:type c:assign c:mkdir c:protect c:execute
```

Chapter 12

Filename Completion

In this chapter we'll take a look at FComp, a WShell utility that provides filename-completion, input key remapping, and iconic (drag-and-drop) operations in the Workbench environment.

FComp is a powerful and flexible tool, and the following sections will describe its many features. However, you don't need to absorb all of the information here in order to enjoy filename-completion, and so you may prefer to install FComp and try it out before studying the details.

12-1 Filename Completion

Line editing and command history make it easy to find and modify existing commands, but sometimes there's no way to avoid typing a long and complex filename. Or is there? Filename-completion, in which FComp generates a complete filename based on the partial clues you type, provides a happy solution to the problem.

Here's how it works: you enter a partial command up to the point where you want FComp to provide a filename, and then press ESC or another "completion" key. FComp examines the command line to determine what directories the file must reside in and then creates a list of candidate filenames. If only a single qualified filename exists, FComp inserts the name on the command line at the cursor location and awaits your next request. Otherwise, it inserts as many characters as it can without introducing any ambiguities, and then "beeps" the screen to let you know that there were multiple possibilities. At this point you can press the completion key again to cycle through the filenames—right on the command line, of course—or you can enter additional characters to make the name unique.

All of this happens very quickly due to the design of FComp's directory caching mechanism; in many cases the possible filenames will be held in memory from a previous request. More importantly, FComp will never override a name entered manually, so if a completion request is taking longer than you expected, you can just continue typing the command.

The Completion Process

The filename completion process begins when you press a key defined as a completion or "hotkey." FComp records the key that was pressed and begins processing the client whose window was active at the time.

The particular key used to initiate the event determines the potential key-specific search path, file pattern, and output format, which are used at various stages of the processing. FComp selects a search path, file pattern, and display format using the hierarchy key:command:default. For example, if the key you hit doesn't have a search path, but the associated command does, then the command-specific path will be used.

Parsing the Command Line

When you request a filename completion, FComp must analyze the command line to determine the context of the filename. It first locates the token containing the cursor, called the *active token*, and splits this into a leading *path* and a *pattern*, either or both of which may be null. For example, suppose that the active token is `df1:working/my^`, where the `^` represents the cursor. Then the path is `df1:working` and the pattern is `my`.

FComp then checks to see whether the active token is the first token on the command line, ignoring any leading “white space” characters. If so, the active token is classified as a *command token*, which affects the search directories to be used in the completion. Otherwise it is considered an *argument token*. All argument tokens are equivalent, regardless of their position in the command.

After classifying the active token FComp can then determine which directories to search for potential completion filenames. If the leading path is *absolute*—that is, if it includes an explicit device reference—only that directory is searched. Otherwise the search includes the current directory and the path directories appropriate for the token. For command tokens the search includes the local path directories plus the directories provided with the PATH command-line argument. For argument tokens the search is normally limited to the current directory, unless a key-specific or command-specific PATH argument was defined.

Absolute Tokens. A special case occurs when the active token contains a colon (:) to the right of the cursor. FComp then considers the text to the left of the cursor as defining a pattern for a device name, and the candidate names are selected from the AmigaDOS device list. For example, the completion of `D^:libs` would consist of all device, directory, and volume names beginning with `D`.

Selecting the Names

Once the search directories have been identified FComp must examine each directory to create a list of the names. If the directory has been cached and its timestamp is current, the filenames are available immediately, but otherwise FComp must examine each directory entry. This is the slowest part of the operation, especially if the directories are on floppy-disk volumes. However, FComp is smart about caching filenames, so it won't hit the disk again unless you modify the directory.

FComp is asynchronous when examining a directory, so other completion requests can be intermingled. Currently it allows only one directory-search adventure at a time, in order to avoid potential thrashing in the filing system.

The next step is to weed out any potential completion names that don't match the pattern supplied on the command line. FComp recognizes characters before and after the cursor as being part of the pattern, so completing `foo ^.c` will expand to filenames ending in `.c`, and `foo myf^` will expand to files beginning with `myf`. The potential completions must also be of the correct type; FComp can tell from the context whether to accept both files and directories or just directories. For example, in the line `cd dh0:/exec` it knows that only a directory name is appropriate.

After the command-line pattern has been applied, the list may be narrowed further by a key- or command-specific pattern. A key or command pattern is applied only to filename completions, not to devices or directory names. The reasoning behind this is as follows: if you're looking for a particular name, the candidate file might exist within a subdirectory, so the subdirectory name should be considered as a completion.

Displaying the Completions

After a qualified list of completion names is available FComp can proceed to display the string. If the result of the search was ambiguous (or unsuccessful), FComp fills in as many characters as it can and then “beeps” the display. This initial display string is called the “match string” and is formally defined as the maximal leading substring of the completion strings. Note that the match string is built from an actual filename in order to preserve the alphabetical case. The match string is displayed using the format `%f%m%l`.

At this point you can enter some more characters and try again, or press ESC (or another completion key) to cycle through the possible completions. The choices are displayed using the default display format is `%f%O%l`, unless a key- or command-specific format was provided, in which case the specific format is used. The SORT and GROUP configuration options control the order in which the choices are presented.

A significant capability of FComp is that it will provide the full pathname for files found along a search path instead of in the current directory. This feature allows FComp to supply a “data path” for any command. Note that the completer supplies the path only when displaying a filename using an output format such as `%f%O%l`; the match string is always displayed without a path.

Unique Completions. FComp treats the case of a unique completion in a special way. It skips the display of the match string (which would have been the entire name anyway) and proceeds to display the string using the chosen key, command, or default output string. This ensures that the displayed name will include a leading path, if appropriate.

In addition, if the key that selected the unique completion has the AUTO attribute, FComp appends a carriage return to the displayed command.

RAW: Mode Operation

Certain applications (such as some text editors) place the console handler in RAW: mode in order to have complete control over the console display. In addition, the application may expect to use the keyboard events that you've defined for filename-completion. Since filename-completion is inappropriate in this situation, special processing is required for completion attempts in RAW: mode.

FComp translates the completion key and qualifier into ASCII characters using the `RawKeyConvert()` function, and then forces the input into the console handler. This makes it appear as though the completion keys weren't being trapped, and the RAW: mode application then works normally.

12-2 Command Line Options

FComp accepts a number of command line arguments to help you tailor it to your system. The `fcomp` command can be issued at any time and will either start the filename-completer—if it's not yet active—or update the command options. The `fcomp` argument template is

`FROM/K,-QUIT/S/B,PATH/K,IGNORE/K,MEMINIT/K,MEMCHECK/K,MEMMAX/K`

and the arguments can be grouped into three categories: (1) `FROM` and `-QUIT` are control options, (2) `PATH` and `IGNORE` are global default strings, and (3) `MEMINIT`, `MEMCHECK`, and `MEMMAX` are operational tuning parameters.

FROM. The `FROM` argument specifies the filename of the FComp configuration file. The default is `S:Config-FComp`. Ordinarily you will not need to specify a `FROM` file if you're using the default configuration file name. However, if FComp is already active and you want to change its configuration, specifying the `FROM` keyword will force it to reread the configuration file.

-QUIT. The `-QUIT` keyword terminates FComp.

PATH. The `PATH` argument supplies a list of directories to be used as an extended command search path. If the filename being completed is the command token—the first word on the line—FComp searches the current directory and the local path directories for possible completions. By default FComp will **not** search `C:`, since it's usually quite large. However, if you want `C:` or other non-path directories to be searched, you can include them as the `PATH` argument when you start FComp. For example, `fcomp PATH df1:c:,sys:system` would add three additional directories to the search path. The usage of `PATH` is analogous to WShell's `ENV:PATH`, so white space, commas, semicolons, and vertical bars are accepted as separators.

IGNORE. This argument supplies a string of characters to be ignored when preceding a filename on the command line. Any character in the string is considered as a separator when the filename is extracted from a command line. The default `IGNORE` string is `><`, which allows redirection characters to butt up against a filename to be completed.

MEMINIT. The `MEMINIT` argument specifies the amount of memory to allocate when the filename-completer is first activated, usually in your **startup-sequence**. The default value is 6,000 bytes.

MEMCHECK. The `MEMCHECK` argument sets the allocation level at which the internal memory-reclaimer becomes active. Once this much memory has been allocated, FComp will attempt to reclaim storage before asking the system for more. Reasonable values for `MEMCHECK` are in the range 25-50 percent of the `MEMMAX` setting, and the default is 10,000 bytes.

MEMMAX. `MEMMAX` is the maximum amount of memory that FComp will allocate. After this point FComp will start dropping names and shedding directories when it runs out of memory, and in a severe shortage it may be unable to accept any new WShell clients. The default value is 40,000 bytes.

12-3 The FComp Configuration File

FComp uses a configuration file to define options, completion keys, keymapping keys, command-specific completions, and icon classes. The configuration file is processed when FComp is first activated or if the `fcomp FROM` command is issued. The default file is called `S:Config-FComp`.

The configuration file consists of **OPTIONS**, **FILETYPE**, **KEY**, and **COMMAND** records, which are identified by their leading keyword. Blank lines are ignored, and comments beginning with a semicolon (;) may be included for readability. You can also add a comment at the end of a record.

FComp displays an error message (with the line number) if an error is detected in the configuration file. Configuration errors are non-fatal, but may result in some keys or commands not working. Note that FComp will not put up a requestor if the configuration file is on an unmounted volume, as this would block all WShells until satisfied.

The OPTIONS Record

The **OPTIONS** record specifies the setting for several FComp options. Its argument template is

`SORT/S, GROUP/S, NOPATH/S, NOTOOLTYPES/S`

and only a single **OPTIONS** record should appear.

SORT. The **SORT** keyword tells FComp to sort the filenames alphabetically before displaying the choices. The default is to present them in the order in which they came from the various directories.

GROUP. The **GROUP** keyword tells FComp to gather the completion names into the categories *devices*, *directories*, and *files* before displaying them. **GROUP** is usually used in conjunction with the **SORT** option.

NOPATH. The **NOPATH** keyword switch suppresses the automatic searching of the client shell's (local) path. You can still supply a **PATH** string with **NOPATH** in effect.

NOTOOLTYPES. The **NOTOOLTYPES** option switch suppresses the reading of tooltypes from an icon file.

The FILETYPE Record

The **FILETYPE** record defines classes of icons and provides the output format to be used for the class. Its argument template is

`FILETYPE/K, FMT/K, REMOVE/S`

and any number of **FILETYPE** records may be defined.

FILETYPE. The **FILETYPE** argument supplies the name of the icon class. This name is checked for a match whenever a **FILETYPE** tooltype is included in an icon.

FMT. The FMT argument specifies the output format to be used when an icon of this FILETYPE class is dropped onto a WShell window.

REMOVE. The REMOVE switch will delete the specified FILETYPE definition.

The KEY Record

KEY records are used to define completion keys or keymapped keys. The argument template is

KEY/K,QUAL/K,NAME/K,PATH/K,PAT/K,FMT/K,AUTO/S,REMOVE/S

and any number of keys can be defined.

KEY. The KEY argument is the keycode of the selected key as a decimal number. The keycodes for the Amiga's keyboard are given in the example file :FComp/Config-Keymap.

QUAL. The QUAL argument is the qualifier code for the selected key as a decimal number. Qualifiers can be specified for the SHIFT, CTRL, ALT, and AMIGA keys.

NAME. The NAME keyword identifies as the record as a keymap definition rather than a completion key. When the key is pressed, FComp sends the FMT format string as forced input to the console. No completion actions are involved, as this is simply a keymap mechanism. The PATH and PAT arguments are ignored for keymapped keys.

PATH. The PATH keyword provides a list of directories to be used as a key-specific argument search path. As in the case of the command-level PATH, the directories can be separated by a comma, semicolon, or white space.

PAT. The PAT keyword supplies a key-specific file pattern, which must be of the form xxx#?yyy. The pattern is used to narrow the list of potential filenames before being displayed.

FMT. The FMT keyword supplies a key-specific output format. This format is used in place of the default whenever the particular key initiates the completion display.

AUTO. The AUTO option on a hotkey will append a RETURN if the completion is unique. This option is potentially dangerous, so use it with care!

REMOVE. The REMOVE option instructs FComp to delete the particular key definition. This option should be required only infrequently.

Examples:

```
KEY 29 FMT ";Choices: %0 %1 %2 %3 %4 %5 %6 %7 %8 %9"
KEY 34 QUAL 16 PATH usr:doc PAT #?.doc FMT %0
KEY 62 FMT "%f%0%1" AUTO ; Achtung! ... auto-return on numeric pad 8
KEY 76 QUAL 16 NAME LALT-UARROW FMT "*E[5~" ; search up
```

The COMMAND Record

The **COMMAND** record allows you to customize the filename-completer for a particular command, possibly to give it a specific search path, file pattern, or output format. Whenever FComp detects that the requested completion applies to a custom command, it uses the command-specific values. The argument template for the **COMMAND** record is

COMMAND/K,PATH/K,PAT/K,FMT/K,AUTO/S,REMOVE/S

and all arguments except for **COMMAND** itself are identical to the **KEY** record.

COMMAND. The **COMMAND** argument provides the name of the command context. FComp uses the WShell alias command-abbreviation algorithm, so you can specify the permissible abbreviations for the commands. For example, **COMMAND EXEcute** would recognize **exe** as **execute**, but not **ex**.

There are actually two different situations in which command-specific completions may be useful. In the more obvious case you would enter a command name and request filename completion for the argument. For example, in the command **execute start^** you want to complete the command with a script file beginning with **start**. Since **execute** looks for scripts in both the current directory and **S:**, it could be given **S:** as a **PATH** argument.

The more subtle use for command-sensitive completions arises when an active program prompts for an input line. Since the input line won't directly provide the command name, FComp must infer the appropriate command context by observing that a command is loaded and that it has requested an input line. Both WShell and the console handler cooperate in this effort.

12-4 Output Formats

FComp provides a flexible method of specifying output strings by using a format interpreter. The format interpreter is similar in design to the one used by WShell for prompt strings and the window titlebar, but uses format tokens more appropriate for filename completion.

Output formats may be defined as either key-specific or command-specific. Formats are selected following the usual key:command:default hierarchy, so that a key format will override a command format, and either will override the default. The current output format tokens are:

Table 12.1 Output Format Tokens

Token	Action
%c	Position the Cursor
%f	First Part of Command Line
%l	Last Part of Command Line
%m	Maximal Leading Substring
%0-%9	Completion Filenames
%-	Rotate Backwards

The completion list is automatically rotated forwards after every display, so that **%1** becomes **%0**, **%2** becomes **%1**, and so on. To rotate the list backwards, you'll can place a **%-%** in the format string.

Ordinary text following one of the completion names is contingent on the existence of that completion, up through the next format specifier. For example, if you specify a format `%1 Does this file exist?%0`, you'll see the question text only if at least two completions exist.

Format Modifiers. The completion filenames `%0-%9` can be modified by preceding the index with one of the “projections” `e`, `h`, `r`, or `t`, following the usage in the popular C Shell. The example in Table 12.2 below shows the various modified names, assuming that the `%0` filename is `SOURCE:files/myfile.c`.

Table 12.2 Filename Modifiers		
Modifier	Extracted Value	Example Value
<code>%e</code>	File Extension	<code>.c</code>
<code>%h</code>	Head Name	<code>SOURCE:files</code>
<code>%r</code>	Root Name	<code>SOURCE:files/myfile</code>
<code>%t</code>	Tail Name	<code>myfile</code>

You can use these modifiers to transform a completion name into a derived or associated filename. For example, `%r0.o` would turn `df1:myfile.c` into `df1:myfile.o`.

12-5 Keymapping with FComp

In addition to its duties as a filename-completer, FComp can also act as a keymapping utility for your WShell windows. This extra functionality was easy to add since the input event handler to detect keystrokes was already in place; the only additional requirement was to specify that certain keys are simply keymapped rather than acting as a completion request.

Keymap keys are specified by supplying a **NAME** argument for the key definition in the configuration file. It doesn't matter what **NAME** you provide—even the null string "" will work—but you may want to describe the key to make the file more readable.

To remap a key, just include the keycode and qualifier in the configuration file, along with the string giving the keymapped value, and provide a **NAME** string. Then when you press that particular key, FComp will insert the string equivalent just as though it had been entered directly. The string can consist of both text and escape-sequences to instruct the console handler to take specific actions. For example,

```
KEY 76 QUAL 16 NAME LALT-UARROW FMT "*E[5~"
```

defines the left-ALT up-arrow key (keycode 76 with qualifier 16) to return the escape-sequence `"*E[5~"`. This particular sequence tells the console handler to search backwards through the command history.

The WShell distribution disk contains an example file called **Config-Keymap** to help you create your own keymap definitions. It contains keycode values for all of the Amiga's keys and gives examples of passing escape sequences to the console handler.

12-6 WShell in the Workbench Environment

AmigaDOS 2.0 provides a new facility with its Workbench environment called *Application Windows*, or AppWindows for short. This allows a software application to register its windows with Workbench in order to receive notification messages when the user moves icons into the window area. The AppWindow facility extends the actions associated with icon manipulation beyond just the Workbench windows, and effectively integrates the external application into the Workbench.

Since WShell often operates with one or more windows on the Workbench screen, it makes a good candidate for AppWindow registration. The natural action of “dropping” an icon on the shell window is taken to mean posting the icon’s filename to the shell’s command line. With the addition of a few tooltype conventions, the icon can customize the text posted to the command, and can even arrange for an automatic RETURN to run the command.

AppWindow registration is managed by FComp, as it already has the mechanisms in place to track the shell’s window and post text to the command line. The registration is automatic and transparent; whenever Workbench is running, FComp registers the WShell windows as AppWindows. Note that AppWindows are supported only for AmigaDOS 2.0 systems.

The text appended to the command line by dropping an icon is treated as normal input text, and uses the FComp output formatting conventions. The default format is %a, in which the %a represents the icon’s filename. Multiple icons using Workbench’s extended selection can be dropped as a group, and each icon name is appended to the command line.

AppWindow Tooltype Conventions

FComp allows you to use tooltype parameters to customize the text strings generated when an icon is dropped into a WShell window. FComp recognizes two tooltypes in the icons: WSHAPP and FILETYPE. Other tooltypes may be present, but are ignored by FComp.

WSHAPP Tooltype. The WSHAPP tooltype directly supplies a format string to be used in place of the default, and the string obeys the usual AmigaDOS quoting conventions.

For example, a tooltype WSHAPP="list %a *N" will run the list command with the icon’s filename as an argument, and the *N specifies an automatic “newline” for the command. Thus the expanded command will run immediately after the icon is released.

FILETYPE Tooltype. The FILETYPE tooltype is somewhat more subtle in its action. The tooltype value is used as a generic class or type for the icon, and that type name is matched against the list of known FILETYPE values specified in the FComp configuration file. If a matching filetype is found, the format string for that filetype is then used to expand the text on the command line. This allows you to specify a set of default actions for different classes of icons.

For example, suppose that the icon contained the tooltype `FILETYPE=ILBM` and that the configuration file contained a `FILETYPE ILBM FMT="view %a"` record. Then the format `view %a` would be used to expand the icon's filename.

12-7 Caching and Memory Management

Intelligent directory caching is vital to the efficient operation of the filename-completer. FComp keeps the names of the files in a directory for as long as possible, under the assumption that if you needed them once, you'll probably need them again. Each time you reference the directory, FComp checks whether the timestamp for the directory matches the cached timestamp value. If it does, the directory hasn't been changed since the last time it was examined and the names are therefore still valid.

If memory space were unlimited, the caching would be easy. However, FComp assumes that you have better uses for your memory space than holding all possible filenames, so it implements provisions for recycling its internal memory. This means that sometimes it will have to go back and reexamine a directory that has been previously used, but this should be an infrequent occurrence.

The algorithm for reclaiming memory is reasonably smart. It defines a class of least-recently-used directories and estimates the amount of time that will be lost by freeing each directory, and then releases memory on a minimum-lost-time-per-byte-reclaimed basis. It can also free up completion strings held for a client if it notices that the client has (manually) changed the command line.

Memory management also helps reduce memory fragmentation. When several programs are simultaneously active and need to allocate memory, the available system memory tends to get broken up into small pieces. FComp reduces this problem by making less-frequent requests for larger blocks of memory and then reusing them whenever possible.

FComp provides three command-line arguments to tune the caching algorithm: `MEMINIT`, `MEMCHECK`, and `MEMMAX`. `MEMINIT` is the amount of memory to be preallocated when the filename-completer first starts up. `MEMINIT` helps reduce memory fragmentation by letting you specify an amount in advance. Since FComp is usually activated shortly after the computer is booted, it is better to obtain the required memory at an early stage.

`MEMCHECK` sets the allocation level at which the internal memory-reclaimer becomes active. Until this much memory has been allocated, FComp simply requests more when it needs more. However, once the `MEMCHECK` limit has been allocated, FComp will attempt to reclaim storage before asking for more from the system.

`MEMMAX` is the maximum amount that FComp will allocate. After this limit has been reached FComp may be unable to accept new client shells and may ignore some potential completion names.

Chapter 13

Using the PathHandler

The concept of a search path for program or data files recurs frequently in software applications. Traditionally the searching capability has been implemented as part of the application itself. For example, the Amiga command shell provides an explicit search path for commands, complete with a special-purpose command (`path`) to maintain it. This works well to the extent that the original design anticipated the need for search paths, but changing patterns of usage may later render the application ineffective or inconvenient.

The PathHandler was conceived as a mechanism to provide a path-searching facility in a manner completely transparent to the application. The PathHandler is implemented as a trick AmigaDOS handler that simulates a concatenation of directories, thereby allowing you to combine several different directories into a single “path” directory. The internal logic of the handler conducts a sequential search of the path directories and thus provides a transparent search facility for any software application. Since the concatenation is done at the DOS level, any software that uses normal DOS calls will work with the PathHandler. In particular, system directories like `LIBS:` or `FONTS:` can be spread over several directories and can even span multiple volumes.

Paths may refer to any other devices, directories, or volumes, including other path directories. Devices referenced in a path need not be mounted or created at the time the path is defined, but will be requested when actually needed for a search. Directories included in a path can be redefined at any time, and the path will use the definition in effect when the search takes place.

Path directories are created automatically by an attempt to lock an object relative to `PATH:`, as with the `assign` command. By default a path is transient and disappears as soon as the lock is released, though permanent path directories can be created with an `ACTION_CREATE_DIR` packet.

13-1 Defining a Path

The PathHandler defines a path as a list of directories separated by the delimiters comma (`,`), semicolon (`;`), vertical bar (`|`), or white space. Each directory name must include a device specification, as there is no “current directory” for the handler itself. A path can include nested references to other paths up to 10 levels deep. The following are examples of valid path specifications:

```
path:df1:c,df0:devs/printers,jd0:
path:vols:,vols:c
```

When a path is referenced by a call to DOS, any nested definitions are fully expanded into their component directories. Within a path definition a directory name following a device name is distributed to all of the components. For example, if `vols:` had been assigned to `path:df0:,df1:`, then a reference to `vols:c` inside a path would expand to `df0:c,df1:c`. Before adding a new directory to the expanded list, the PathHandler verifies

that the exact specification hasn't occurred earlier in the list, in order to avoid duplicating the search.

A path can be either transient or permanent. Transient paths come into existence whenever a lock on an object relative to the `PATH:` device is requested, most conveniently by using the `assign` command. The transient path persists until all locks on it are released.

Permanent paths can be created by using the AmigaDOS `ACTION.CREATE_DIR` packet from a program. Normally the `makedir` command could be used to create a new (path) directory, but it doesn't work in this case; `makedir` tests whether the directory already exists by attempting to get a lock on it, and then returns an error message when the lock succeeds. Once created, a permanent path can be removed using the `delete` command, provided that there are no outstanding locks on it.

You can display the currently-defined paths by listing `PATH:` (or its volume name `Tao:.`) The listed names are prepended with the `Tao:` name to ensure that any subsequent references to the path are sent to the `PathHandler` device. In addition, a `!` path termination character is appended to the name; this allows the `PathHandler` to determine the actual end of the path in case another application appends a filename to the path.

13-2 Path Aliases

Since the full specification of a path name involves multiple device names with their attendant colons, the path may confuse certain software products that expect to see only a single colon in a file name. In addition, some path names may become quite long, which can be a problem for some "file requestor" routines. To avoid these concerns, the `PathHandler` supports the concept of *alias names* for paths.

A path alias—not to be confused with the aliases used by a command shell—is an alternative name for a path that obeys the normal naming rules for a file or directory. In particular, an alias name will never contain a colon `:` or slash `/` character. Whenever an alias name relative to `PATH:` is found, the alias is expanded into its full path before being processed the `PathHandler`. For example, suppose that `path1` was an alias for the path `DF0:,DF1:.` Then a reference to `PATH:path1` would be expanded to `PATH:DF0:,DF1:.` The `PathHandler` can always tell the difference between a path name and an alias name, since the former must contain at least one colon and the latter will never contain a colon.

Path aliases can be created explicitly by using the `filenote` command to assign the desired alias name as a "comment". Once this is done, the `PathHandler` inverts the normal usage of "name" and "comment" and thereafter reports the alias name as the "name," and the full path as the "comment." The `filenote` command can be used subsequently to effectively rename either the path itself or the alias.

The example below shows the `filenote` command creating a path alias and then modifying the path definition.

```

1> assign foo: path:devs:,env:
1> filenote foo: first
1> list tao:
Directory "Tao:" on Sunday 13-Jan-91
first                               Dir ----rwd Today    16:10:59
: devs:,env:
1 directory - 1 block used

1> filenote foo: "devs:,l:" ; change the path definition
1> list tao:
Directory "Tao:" on Sunday 13-Jan-91
first                               Dir ----rwd Today    16:10:59
: devs:,l:
1 directory - 1 block used

```

The use of a path alias also allows the AmigaDOS 2.0 Workbench "Info" tool to modify the path definition. With the Tao: icon set for "Show All Files" and "View by File," you can attach an alias name to a PathHandler definition and then modify the path by changing the comment field in the "Info" display.

Implicit Paths

Path aliases can also be recognized as implicit references to a device or assigned directory, which gives the PathHandler the ability to refer to any device or volume even if a path to it hasn't been previously created. Whenever a name relative to PATH: doesn't include a colon and so can't be a full path name, the relative part is taken as an *implicit alias* name. For example, in the name PATH:include/exec, the relative part include/exec is assumed to be a path alias. If an alias name include has been defined, the associated full path is substituted. Otherwise, the name include is checked against the DOS device list, and if it exists there is transformed to be relative to that name. In the above example, if INCLUDE: existed as an assigned directory, then PATH:include/exec would be transformed to INCLUDE:exec.

13-3 Protection Attributes

Path directories can be assigned protection attributes with the protect command, and these attributes are interpreted by the PathHandler when searching a path. If the action to be performed requires a specific permission, the search skips over any candidate directories lacking the necessary attributes. For example, suppose that the following paths are created:

```

1> ASSIGN p1: path:dh0:,df0:
1> PROTECT p1: r
1> ASSIGN p2: path:p1:c,df1:

```

Then a search of P2: requiring only read permission—such as a list command—will search the P1: directories, but a search requiring delete permission will go directly to the DF1: node.

The semantics associated with the **r** (read), **w** (write), and **d** (delete) attributes follow their canonical meanings. Read permission is required to open a file for reading or to list a directory's contents, write permission is required for writing or updating a file, and delete permission is necessary to delete a file from a directory. However, the **e** attribute, traditionally meaning "executable," is interpreted by the PathHandler as "expand" permission. When a directory's contents are being listed (e.g. with the **list** or **dir** commands), the **e** attribute determines whether the component directories will be entered. The listing below illustrates the effect of the **e** attribute:

```
1> assign NewC: path:c:,work:c
1> protect NewC: -e
1> list NewC:
Directory "Tao:Tao:c:,work:c" on Saturday 09-Mar-91
c:                Dir ----rw-d Today      16:01:26
work:c            Dir ----rw-d Today      16:01:26
2 directories - 2 blocks used
```

Instead of expanding the **c:** and **work:c** directories into their component files, only the component directories of the path itself are shown. This results in faster operation for large directories.

Note that the protection attributes used are those of the PathHandler entries and not the component directories themselves. This allows you to define several paths containing the same component directories but with different associated protections.

13-4 Directory Timestamping

To improve its operation with utilities sensitive to timestamps, the PathHandler will update its directory timestamps after any operation that would have updated the timestamp of an external directory contained in the path. Following the successful completion of a **CreateDir** (**makedir**), **Delete**, **FindOutput** (open in mode **NEWFILE**) and **Rename** packet, the path directory timestamp is set to the current time.

Directory timestamps may also be updated during a directory scan, at which time the timestamp is set to the later of its current value and the timestamps of the scanned objects.

13-5 Technical Details

The PathHandler employs a number of tricks in order to provide a consistent implementation of the AmigaDOS packets it supports. Two packet types, **ACTION_FINDOUTPUT** and **ACTION_RENAME**, presented special problems in implementation, and are therefore explained in greater detail here.

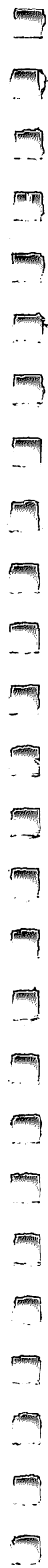
An **ACTION_FINDOUTPUT** packet is generated whenever a program attempts to open a new file for writing. For example, an editor writing a file out to disk will usually open a file in this mode. The simple implementation of this packet in the PathHandler would have been to walk the path directories until one succeeded in opening the file, and then terminate the search. However, often the file being opened for writing will have been previously read from an existing file, and in most cases the new file should be written back over the old

file. The simple implementation would instead always place the file in the first writable directory in the path.

The “write-back” feature is supported by having the PathHandler walk the path list first attempting an `ACTION_FINDINPUT` packet (the “read file” equivalent). If this operation succeeds, the packet is transformed to an `ACTION_END` to close the opened filehandle, and then changed back to an `ACTION_FINDOUTPUT` with the path index set at the point where the file was found. If the `ACTION_FINDINPUT` or second `ACTION_FINDOUTPUT` attempt fails, then the path index is reset and the entire operation repeated from the beginning as an `ACTION_FINDOUTPUT`.

Supporting the `ACTION_RENAME` packet similarly requires several steps, but is even more complex because two separate paths are involved — one for the old name of the file, and another for the new name. The rename operation begins by walking the old name path with an `ACTION_LOCATE_OBJECT` (lock) packet to attempt to find the named file. If this succeeds, the packet is transformed back into an `ACTION_RENAME`, and the PathHandler then walks the second (new name) path until this packet succeeds or the path is exhausted.

Two possible failures can occur: if the `ACTION_LOCATE_OBJECT` packet fails, the operation terminates with an error code of 205 (“object not found.”) If the `ACTION_RENAME` packet fails, the final error is reported as an error 215 (“rename across devices.”)



Appendix A

DisplayHandler Control Sequences

The DisplayHandler provides an escape-sequence equivalent for all of its internal editing and control features, thereby allowing an external application to control or remap the handler's features by inserting the appropriate sequences into the read stream. For example, the DisplayHandler menu actions can be created by binding one or more escape sequences to the desired menu item. Another application would be an AmigaDOS 2.0 Commodities Exchange program to intercept and modify the read stream to emulate a preferred line editor.

The DisplayHandler escape sequences are of the form "<ESC>[nnn]" or "<CSI>nnn", where <ESC> is hex 1B, <CSI> is hex 9B, and nnn is the numeric code in ASCII digits. The control keys use codes 32-63, alt-control keys use codes 64-95, and the function keys use codes 96-115. For example, the code for ALT-CTRL-A (in hex \$81) would be 64+(\$81-\$80), giving the escape sequence of "<ESC>[65]".

Codes 16-31 are used for the arrow keys and various special sequences. Codes 0-15 are reserved for the system control sequences issued by the Amiga's console.device and should not be used. For example, the close gadget event uses the console-device raw-event sequence coding of "<ESC>[11]".

The escape sequences and their associated actions are listed in the table below.

Table A.1 DisplayHandler Editing Keys

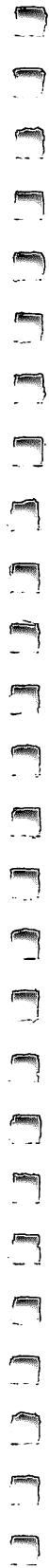
Escape Sequence	Default Key	Action
<ESC>[16]	Up-Arrow	Previous History Line
<ESC>[17]	Down-Arrow	Next History Line
<ESC>[18]	Right-Arrow	Cursor Right
<ESC>[19]	Left-Arrow	Cursor Left
<ESC>[20]	Shift-Up-Arrow	First History Line
<ESC>[21]	Shift-Down-Arrow	Last History Line
<ESC>[22]	Shift-Right-Arrow	Cursor Right Word
<ESC>[23]	Shift-Left-Arrow	Cursor Left Word
<ESC>[24]	Shift-Tab	(unassigned)
<ESC>[25]	DEL	Delete Character
<ESC>[26]	Help	(unassigned)
<ESC>[27]	(none)	Window Bounds Report
<ESC>[28]	(none)	Cursor Position
<ESC>[29]	(none)	Paste Request
<ESC>[30]	(none)	Insert Character
<ESC>[31]	(none)	(unassigned)

Table A.1 (cont.) DisplayHandler Editing Keys

Escape Sequence	Default Key	Action
<ESC>[32]	CTRL-@	Set Mark
<ESC>[33]	CTRL-A	Toggle Insert/Overstrike
<ESC>[34]	CTRL-B	Clear History Buffer
<ESC>[35]	CTRL-C	Control-C Break Signal
<ESC>[36]	CTRL-D	Control-D Break Signal
<ESC>[37]	CTRL-E	Control-E Break Signal
<ESC>[38]	CTRL-F	Control-F Break Signal
<ESC>[39]	CTRL-G	Bell (beep screen)
<ESC>[40]	CTRL-H	Backspace
<ESC>[41]	CTRL-I	Tab
<ESC>[42]	CTRL-J	Newline
<ESC>[43]	CTRL-K	Cut to EOL to Save Buffer
<ESC>[44]	CTRL-L	Formfeed
<ESC>[45]	CTRL-M	RETURN
<ESC>[46]	CTRL-N	Alternate Shift In
<ESC>[47]	CTRL-O	Alternate Shift Out
<ESC>[48]	CTRL-P	Insert from Save Buffer
<ESC>[49]	CTRL-Q	Release Output
<ESC>[50]	CTRL-R	Recall Typed-Ahead Line
<ESC>[51]	CTRL-S	Hold Output
<ESC>[52]	CTRL-T	Transpose Characters
<ESC>[53]	CTRL-U	Delete to Beginning
<ESC>[54]	CTRL-V	Paste from Clipboard
<ESC>[55]	CTRL-W	Refresh Display
<ESC>[56]	CTRL-X	Delete Line
<ESC>[57]	CTRL-Y	Delete to End
<ESC>[58]	CTRL-Z	Flush Typed-Ahead Lines
<ESC>[59]	CTRL-[	ESC
<ESC>[60]	CTRL-\	End-of-File
<ESC>[61]	CTRL-]	Toggle Start/End Line
<ESC>[62]	CTRL-^	Enter Insert Mode
<ESC>[63]	CTRL-_	Load from "Undo" Buffer
<ESC>[64]	ALT-CTRL-@	Delete to Beginning
<ESC>[65]	ALT-CTRL-A	Copy Session to Clipboard
<ESC>[66]	ALT-CTRL-B	Clear Session
<ESC>[67]	ALT-CTRL-C	(unassigned)
<ESC>[68]	ALT-CTRL-D	(unassigned)
<ESC>[69]	ALT-CTRL-E	Copy Page to Clipboard
<ESC>[70]	ALT-CTRL-F	Copy Line to Clipboard
<ESC>[71]	ALT-CTRL-G	(unassigned)
<ESC>[72]	ALT-CTRL-H	Delete Preceding Word
<ESC>[73]	ALT-CTRL-I	Skip Forward Name
<ESC>[74]	ALT-CTRL-J	(unassigned)
<ESC>[75]	ALT-CTRL-K	Delete Forward Word

Table A.1 (cont.) DisplayHandler Editing Keys

Escape Sequence	Default Key	Action
<ESC>[76]	ALT-CTRL-L	(unassigned)
<ESC>[77]	ALT-CTRL-M	(unassigned)
<ESC>[78]	ALT-CTRL-N	(unassigned)
<ESC>[79]	ALT-CTRL-O	Skip Back Name
<ESC>[80]	ALT-CTRL-P	(unassigned)
<ESC>[81]	ALT-CTRL-Q	(unassigned)
<ESC>[82]	ALT-CTRL-R	(unassigned)
<ESC>[83]	ALT-CTRL-S	(unassigned)
<ESC>[84]	ALT-CTRL-T	(unassigned)
<ESC>[85]	ALT-CTRL-U	Delete Back Name
<ESC>[86]	ALT-CTRL-V	(unassigned)
<ESC>[87]	ALT-CTRL-W	Copy Region
<ESC>[88]	ALT-CTRL-X	Cut Region
<ESC>[89]	ALT-CTRL-Y	Delete Forward Name
<ESC>[90]	ALT-CTRL-Z	(unassigned)
<ESC>[91]	ALT-CTRL-[	CSI
<ESC>[92]	ALT-CTRL-\	(unassigned)
<ESC>[93]	ALT-CTRL-]	(unassigned)
<ESC>[94]	ALT-CTRL-^	(unassigned)
<ESC>[95]	ALT-CTRL-_	(unassigned)
<ESC>[96]	F1	Shrink Window (toggle)
<ESC>[97]	F2	Zoom Window (toggle)
<ESC>[98]	F3	Bottom of Session
<ESC>[99]	F4	Top of Session
<ESC>[100]	F5	Search Down
<ESC>[101]	F6	Search Up
<ESC>[102]	F7	Session Line Down
<ESC>[103]	F8	Session Line Up
<ESC>[104]	F9	Screen Front/Back
<ESC>[105]	F10	Window Front/Back
<ESC>[106]	Shift-F1	(unassigned)
<ESC>[107]	Shift-F2	(unassigned)
<ESC>[108]	Shift-F3	Skip Left Name
<ESC>[109]	Shift-F4	Skip Right Name
<ESC>[110]	Shift-F5	(unassigned)
<ESC>[111]	Shift-F6	(unassigned)
<ESC>[112]	Shift-F7	Session Page Down
<ESC>[113]	Shift-F8	Session Page Up
<ESC>[114]	Shift-F9	(unassigned)
<ESC>[115]	Shift-F10	(unassigned)



Index

- & (ampersand) character, 11, 23, 85-86
- * (asterisk) character, 7, 12, 21, 85
- ` (backtick) character, 13
- \$ (dollar-sign) character, 13
- " (double-quote) character, 12, 20, 51
- ! (exclamation-mark) character, 11, 98
- [(left-bracket) character, 17, 45, 85
- + (plus-sign) character, 12, 22
- # (pound-sign) character, 27
- ;(semicolon) character, 12, 17, 76, 78, 83, 97
- | (vertical-bar) character, 11, 17, 69, 85
- abbreviations, for commands, 64, 93
- absolute path, 88
- ACTION_FINDOUTPUT packet, 100
- ACTION_QUEUE packet, 72
- ACTION_RENAME packet, 101
- ACTION_SET_COMMENT packet, 67
- ACTION_STACK packet, 72
- ALIAS built-in command, 50, 61-63, 79, 86
 - LIST keyword, 63
 - GLOBAL keyword, 62
 - LITERAL keyword, 61
- ALIAS configuration record, 86
- alias list, 61
- aliases, path 98
- ALT-CTRL-A, editing key, 38
- ALT-CTRL-B, control key, 39
- ALT-CTRL-E, editing key, 38
- ALT-CTRL-F, editing key, 38
- ALT-CTRL-I, editing key, 36
- ALT-CTRL-O, editing key, 36
- ALT-CTRL-W, editing key, 37
- ALT-CTRL-X, editing key, 37
- alternate names, for resident commands, 47
- Amiga-E command, 24
- AmigaDOS, 4-6, 10, 14-16, 20, 24, 27-29, 31, 34, 45-47, 49, 50, 54, 84-85, 94-95, 98
 - RESIDENT command, 45, 47
 - resident list, 16, 45-47
 - Version 2.0, 4-6, 10, 14-16, 20, 24, 27-29, 31, 34, 45-46, 49-50, 54, 94-95, 98
 - 2.0 Shell, 14-15, 84-85
- ampersand character, 11, 23, 85-86
- ANSI escape sequence, 65
- application windows, 94
- AppWindows, 94-95
 - registration, 95
- ARexx, 10, 16-17, 24, 45, 58, 63, 66, 70-71, 75-81
 - server process, 16, 75
- argument substitution, 62-63
 - with inline ARexx program, 63
- argument template,
 - for ALIAS command, 61
 - for ALIAS record, 86
 - for COMMAND record, 93
 - for DHOPTS command, 25
 - for EXECIO command, 71
 - for FCOMP command, 90
 - for FILETYPE record, 91
 - for KEY record, 92
 - for menu items, 33
 - for NEWWSH command, 20
 - for OPTIONS record, 84, 91
 - for RESI command, 46
 - for RESI record, 86
 - for SPECIAL record, 85
- ASSIGN command, 6, 26
 - removing devices, 6, 26
- asterisk character, 7, 12, 21, 85
- AUTO keyword, with RESI command, 47
- AUTO DisplayHandler option, 22, 28
- AUTO FComp option, 89
- auto-loading resident commands, 47
- auto-open window, 21, 28-29, 34
 - with AUTO option, 28
 - with DEACT option, 29
 - with QUIET option, 21
 - with screen jumping, 34
- AUTOPUSH configuration option, 55-56, 59, 84
- background character, 11, 23, 62, 84-85
 - changing value, 23, 85
 - with ONEBACK option, 23
- background commands, 22-23, 85
 - input stream for, 23
- background shell, 19, 22-24, 69, 85
 - creating, 22

- priority for, 19
- receiving break signals, 85
- BACKSPACE**, editing key, 36
- backtick character, 13
- backticked commands, 13-14, 66
 - in prompt string, 66
 - in variable values, 13
- multiple, 14
- blocking, with piping, 69
- break signals, 29, 70, 85
 - CTRL-C from pipe, 70
 - with **BRKMODE** option, 29
 - with **NOBREAK** option, 85
- broken pipe, 70
- built-in commands, 16-17, 45, 47, 49-59
 - AmigaDOS, 49
 - avoiding search for, 17
 - deleting, 47, 49
 - displaying, 49
 - in search order, 16
 - return codes from, 50
- CACHELIM** configuration option, 15, 84
- caching commands, 15, 84
- CD** built-in command, 9, 50, 84
- CHANGED** configuration option, 63, 84
- CHECKICON** configuration option, 10, 59, 84
- checksum, 45
- clearing command history, 40, 42
- clearing session history, 39
- CLI**, 1, 9, 11, 17, 23
 - redirection restriction, 11
- CLI** process, 20
- close gadget, 22, 28-30, 32, 52
 - alternative to **ENDCLI**, 52
 - with **AUTO** option, 28
 - with **CLOSE** option, 29
 - with **NOCLOSE** option, 30
 - with **WAIT** option, 32
- CLOSE** DisplayHandler option, 22
- CMD** tooltype, 22
- CNN**: device, 26
- CNX**: device, 26
- code purity, 15, 45
- command abbreviations, 64, 93
- command alias, 40, 61
- command arguments, 50
 - prompting for, 50
 - substitution of, 62
- command continuation, 12
- command history, 35, 40-41
 - clearing, 40, 42
 - options for, 40
 - preloading, 41
 - setting size, 41
- Command Line Interface (CLI), 1
- COMMAND** records, in FComp configuration, 93
- command abbreviations, 64, 93
- commenting out tooltype, 21
- comments, 12, 83, 91
 - in configuration file, 83
 - in FComp configuration, 91
 - on command line, 12
- compatibility, 1
- CON**: device, 1
- Config-FComp** file, 4, 90-91
- Config-WShell** file, 4, 83
- Config-Keymap** file, 92, 94
- configuration file, 4, 11-12, 46, 69, 83-86, 91-94
 - changing background character, 11, 85
 - changing escape character, 12, 85
 - changing piping character, 69, 85
 - Config-FComp**, 4, 91
 - Config-WShell**, 4, 83
 - errors in records, 83
 - record keywords, 83
 - with FComp, 91-94
 - with filename-completion, 90
 - with resident commands, 46
 - with **SPECARG** option, 12, 84
- ConMan**, 1, 33
- console.device**, 29, 43, 103
- console handler, 1, 35, 89, 94
 - escape sequences for, 94
 - RAW** mode with, 89
- CONSOLE** keyword, with **NEWWSH** command, 20
- CONSOLE** tooltype, 22
- console window specification, 25
- COPY** command, 46
- CTRL-~, editing key, 36
- CTRL-@, editing key, 37
- CTRL-A, editing key, 36
- CTRL-B, control key, 40
- CTRL-C break, with piping, 70
- CTRL-K, editing key, 37

- CTRL-P, editing key, 37
- CTRL-Q, control key, 39
- CTRL-R, control key, 38
- CTRL-S, control key, 39
- CTRL-T, editing key, 38
- CTRL-U, editing key, 37
- CTRL-V, editing key, 38
- CTRL-W, control key, 39
- CTRL-X, editing key, 36
- CTRL-Y, editing key, 37
- CTRL-Z, control key, 38
- CTRL-\, control key, 22, 32, 38, 84
- CTRL-_, editing key, 38
- CTRL-], editing key, 36
- curly braces, 13
- current directory, 9, 15-16, 20, 50, 55-56, 59, 66, 88
 - displaying in prompt, 66
 - in search order, 15-16
 - inheriting, 20
 - with CD, 50
 - with filename completion, 88
 - with implicit CD, 16
 - with POPCD, 55
 - with PUSHCD, 56
 - with SWAPCD, 59
- cursor positioning, 36
- custom shell, 24
- cutting off background shell, 23
- cyclic definitions, 64
- DEACT DisplayHandler option, 27-29
- default attributes, with DisplayHandler, 26
- default console, with NEWWSH, 20
- DEL, editing key, 36
- DELETE command, 46
 - DELETE keyword, with RESI command, 47
- deleting resident commands, 47
- device name, as FComp token, 88
- DHOPTS command, 5-7, 25-26, 33, 70
 - argument template for, 25
 - defining PIP: with, 70
 - deleting menu, 26
 - errors from, 6
 - overriding devices, 26
 - reading menu description, 33
 - replacing menu, 26, 33
 - setting device-level defaults, 26
 - with icon, 7
- DHOPTS icon, 22
- directory timestamp, 88, 96, 100
- DisplayHandler, 20, 25-44, 54, 57, 69, 7.
 - HLINES option, 41
 - options, 20, 25
 - option keywords, 27
 - RESAVE option, 40
- dollar-sign character, 13
- double-quote character, 12, 20, 51
- down-arrow key, 40
- drag gadget, 30
- early termination, in REXX search, 77
- ECHO built-in command, 51
- Echo environment variable, 63, 84
- ELSE built-in command, 51
- ENDCLI built-in command, 22, 52, 80
- ENDIF built-in command, 52
- ENDSKIP built-in command, 52
- ENV: device, 5
- environment variable, 11, 14, 17, 20, 51, 63.
 - 66-67
 - Echo, 14
 - in prompt string, 66
 - NoClobber, 11, 14
 - Path, 14
 - Shellwindow, 14, 20
 - Titlebar, 14, 67
 - Trunc, 63
- EOFLIM configuration option, 22, 84
 - setting EOF limit, 84
- errors, in backticked commands, 13
- ESC character, 12
- ESC key, 87
- escape character, 12, 85
- ESCAPE configuration option, 85
- escape, from search order, 17, 45
- exclamation-mark character, 11, 98
- EXECIO command, 1, 71-73, 77, 79
 - argument template for, 71
 - as filter program, 71
 - searching, 73
 - warning error from, 71
 - with piping, 77
 - with QUOTE option, 72
 - with STEM option, 72
- execute protection flag, 15

- execute script, 75
- Execute()** function, with **SetExecute**, 24
- expand permission, 100
- F3**, control key, 39
- F4**, control key, 39
- F5**, editing key, 40
- F6**, editing key, 40
- F7**, control key, 39
- F8**, control key, 39
- FAILAT** built-in command, 53
- failure level, 15, 53, 65
 - default value, 53
 - in prompt, 65
 - nominal values, 53
- FCOMP** command, 4, 6, 39, 44, 90
 - command-line arguments, 90
 - for keymapping, 39, 44
- file extension, with **REXX** programs, 76
- Filehandle, console window, 23
- filename completion, 4, 44
 - activating 6
- FILENOTE** command, 98
- FILETYPE**, 91,95
 - AppWindow tooltype, 95
 - configuration record, 91
- filter program, 70
- :Filters** directory, 70
- flicker, 67
- flow control, 39
- FORMAT** Command, 18
- fragmentation, 96
- FROM** keyword, with **FCOMP** command, 90
- FROM** tooltype, 22
- global aliases, 62
- GLOBAL** keyword, with **ALIAS** command, 62
- global path, 15, 17
 - in search order, 17
- GROUP** option, with **FComp**, 91
- handler, 4
- handshake protocol, 80
- HEAD** filter, 71
- history buffer, sizing, 41
- HISTORY** command, 41-42, 57
 - NUM** keyword, 41, 57
- HLINES** DisplayHandler option, 41

- host address, for **WShell**, 80
- host command, 75
- hotkey, with filename-completion, 87
- Howdy, 65
- icon, 9
- IF** built-in command, 51-54
- IGNORE** keyword, with **FCOMP** command, 90
- implicit **CD**, 15-16, 76, 84
 - disabling, 16
 - in search order, 15-16
 - with **LATECD** option, 16
 - with **NOIMPCD** option, 16
- INACTIVE** DisplayHandler option, 30
- .info** file, 59, 84
- INFO** tool, 22
- inheritance, local aliases, 62
- inherited values, 19, 20
- inline program, 12, 63, 76, 78-79
 - as alias, 79
 - omitting closing quote, 78
- input event handler, 44
- input stream, 9, 23
 - for background shells, 23
 - with **STDIN** command, 59
- insert, editing mode, 35
- Install-WShell** file, 3
- installation, 3
- interactive shell, 19
- ISREXX** command, 66
- JUMP** built-in command, 34, 54
- KEY** records, in **FComp** configuration, 92
- keycode, 42, 94
- keymap editor, 43
- KEYMAP** DisplayHandler option, 44
- keymapping, 39, 42-43, 92, 94
 - defining keys, 92, 94
 - with **FCOMP**, 39, 44
 - with **KEYMAP** option, 30, 43
- keywords, in **WShell** configuration, 83
- LAB** built-in command, 55
- language standard, 80
- LATECD** configuration option, 84
- left-bracket character, 17, 45, 85
- library, 1, 3

line editing, 1, 35
LIST command, 46
 -**LIST** keyword, 48, 63
 with **ALIAS** command, 63
 with **RESI** command, 48
 load-on-call, 45, 47
LOADLIB command, 3, 22
 icon for, 22
 local path, 15, 17, 88, 90
 in search order, 16
 local variables, 12, 14, 17, 20, 63, 66
 in prompt string, 66
 Lock, 17

Make utility, 2
MEMCHECK keyword, with **FCOMP** command,
 90, 96
MEMINIT keyword, with **FCOMP** command, 90,
 96
MEMMAX keyword, with **FCOMP** command, 90,
 96
 memory fragmentation, 96
 menus, 30, 33-34
 description file, 33-34
 MENU option, 30
 message port, 80
MOUNT command, 6
MOUNTED built-in command, 55
 mountlist entry, for **PathHandler**, 4

NAME argument, for keymapping, 94
NAME tooltype, 22
 nested paths, with **PathHandler**, 97
NEWCON: device, 1
 newline character, 12, 17, 51, 65
NEWSHELL command, 24
NEWSH command, 7, 20-21, 62
 CMD keyword, 21
 COMMAND keyword, 21
 FROM keyword, 20
 NAME keyword, 21
 WShell-Startup file, 21
NEWSH icon, 7, 22
 CMD tooltype, 22
 CONSOLE tooltype, 22
 FROM tooltype, 22
NIL: stream, 23
 as "cutoff", 23

NOBATCH configuration option, 84
NOBREAK configuration option, 85
NOCLEAR keyword, with **HISTORY** command,
 42
NoClobber environment variable, 11, 14
NOECHO keyword, with **ALIAS** command, 61,
 63
NOIMPCD configuration option, 85
NONBLOCK **DisplayHandler** option, 30, 39
NOPATH option, with **FComp**, 91
NUM keyword, with **HISTORY** command, 41

ONEBACK configuration option, 84-85
OPTIONS configuration record, 84
 ordering resident list, 46
 output stream, 9, 23
OVER **DisplayHandler** option, 30, 36
 overstrike, editing mode, 30, 32, 36

 parent task, 19, 23
 password security, 40
PATH command, 17
Path environment variable, 14, 17, 51, 90
PATH keyword, with **FCOMP** command, 90
 path, with **PathHandler**, 97
PathHandler, 4, 97-101
 path aliases, 98
 implicit paths, 99
 pathname, 84, 89
 for loaded command, 84
 with **FComp**, 89
PATHNAME configuration option, 84
Path-Mountlist file, 4
 patterns, 50, 69, 88
 FComp command-specific, 88
 FComp key-specific, 88
 with **CD** command, 50
PIP: device, 26, 69-70
 pipe handler, 26, 69-70
 PIP: device, 26, 69-70
 PIPE: device, 69-70
 piping character, 62, 85
PIPING configuration option, 69, 85
 piping, 11, 23-24, 59, 69, 77, 85
 to **STDIN** command, 59
 with **RUNWSH** command, 23
POPCD built-in command, 55, 79
 postponing redirection, 11

- pound-sign character, 27
- preloading command history, 41
- previous versions, use of **Startup-WShell**, 5
- priority, 19, 24, 69
 - for piped shell, 24, 69
- process, 20
- PROGDIR**: directory, 10
- Project, Workbench menu, 22
- prompt, 56, 65-67
- prompt keywords, 65
- prompt programs, 66
- PROMPT** built-in command, 56, 65
- proportional gadget, 6, 30, 39
 - default placement, 30
 - with **LEFT** option, 30
 - with **NOPROP** option, 30
- PROTECT** command, 99
- Public Screens, 31, 34, 54
 - with screen jumping, 34
- pure code, 45
- PUSHC** built-in command, 56
- qualifier key, 43
- qualifier, 89, 92, 94
- QUIET** keyword, with **NEWWSH** command, 21
- QUIT** keyword, with **FCOMP** command, 90
- QUIT** built-in command, 57
- quoting convention, for inline macros, 78
- RAM disk, 45
- RAW**: mode, 31, 89
 - with filename-completion, 89
- RawKeyConvert()** function, 89
- reader program, with piping, 69
- RECALL** built-in command, 57
- redirection, 9, 11, 16, 51, 78
 - postponing, 11
 - restriction with **CLI**, 11
 - with implicit **CD**, 16
 - with inline program, 78
- requestor, 15
- RESAVE DisplayHandler** option, 40
- RESI** built-in command, 16, 46-49, 57, 86
 - argument template, 46
 - AS** keyword, 47
 - AUTO** keyword, 47
 - DELETE** keyword, 47
 - IGNORE** keyword, 47

- LIST** keyword, 48
 - replacing modules, 47
- RESI** configuration record, 86
- resident commands, 15-17, 45-48, 76
 - in search order, 15, 45
- resident list, 15, 45-47, 49, 57
 - AmigaDOS**, 15, 45
 - in search order, 45
 - ordering, 46
 - WShell** private, 15
- return code, 15, 50, 53, 65
- RETURN** key, 34, 36, 40
- :Rexx** directory, 76
- REXX** built-in command, 58, 76, 78
- REXX** macro programs, 16, 70, 75-76
 - in search order, 16
- REXXMAST** command, 49, 76
- ROM Kernel Manual**, 43
- rotation, of completion list, 93
- RUN** command, 11, 22-24
- RUNBACK** configuration option, 23, 85
- RUNWSH** command, 22-23, 62
- RXC** command, 76
- SAVE** keyword, with **HISTORY** command, 41
- screen jumping, 34, 54
- screens, changing mode, 54
- script file, 3-4, 15, 57, 75
- script language, 75
- scrollbar, 6, 30, 32, 38-39
 - with **NOPROP** option, 30
- search keys, 40
- SEARCH** configuration option, 18, 85
- search order, 15
- search path, 14-16, 45, 76-77, 87, 97
 - for resident commands, 45
 - for **REXX** programs, 76-77
 - with filename-completion, 87
- search-escape character, 17-18, 45, 85
 - changing default, 18, 85
- searching, **DisplayHandler** features, 40
- Seglist splitting, 49
- semicolon character, 12, 17, 76, 78, 83, 97
- serial.device**, 26, 29, 31
- session history, 6, 31, 35, 38-39, 54
 - clearing, 39
 - copying to Clipboard, 38
 - sizing, 31

- with **SESSION** option, 31
- with **SPILL** option, 31
- SET** command, with local variables, 14
- SETENV** command, with environment variables, 14
- SetExecute** utility, 24
- SetMap** command, 30, 43-44
 - with **KEYMAP** option, 30
- ShellWindow** environment variable, 14, 20
- shift-down-arrow, editing key, 40
- SHIFT-F7**, control key, 39
- SHIFT-F8**, control key, 39
- shift-up-arrow, editing key, 40
- short path, with **SHORTCD** option, 85
- SHORTCD** configuration option, 85
- simple-refresh, 31
- SKIP** built-in command, 58
- smart-refresh, 31
- SORT** option, with **FComp**, 91
- SPECARG** configuration option, 12, 84
- SPECIAL** configuration record, 85
- spill file, 31-32
- STACK** built-in command, 10, 59
- stack, 9, 58-59, 84
 - default size, 9, 58
 - using icon stack, 84
- Startup-Sequence** file, 4-6, 76
- Startup-WShell** file, old name, 5
- STDIN** built-in command, 59, 70
- stem variables, with **ExecIO**, 72-73, 77
- STICKY** **DisplayHandler** option, 36
- SWAPCD** built-in command, 59
- swapping history, 42
- Tao:**, **PathHandler** volume name, 98
- task name, with **NEWWSH**, 21
- task priority, *see* priority
- task, 19
- TEE** filter, 71
- template, 50
- ticks, 54
- timestamp, directory, 88, 96, 100
- titlebar prompt, 67
- Titlebar** environment variable, 14, 67

- tooltipe, 7, 22, 91, 95-96
 - with **AppWindows**, 95
 - with **FILETYPE** record, 91, 95-96
 - with **NewWSH** icon, 22
 - with **WSHAPP**, 95
 - with **WShell** icons, 7
- true history, history option, 40
- TRUNC** keyword, with **ALIAS** command, 63
- Trunc** environment variable, 63
- truncation, alias display, 63
- unique completion, 89
- unmounted volume, 15, 17
- up-arrow key, 40-41
- User-Startup** file, 5-6
- UserShell** facility, 24
- variable expansion, 13, 66
 - in prompt string, 66
- vertical-bar character, 11, 17, 69, 85
 - with piping, 69
- WBStartup** drawer, 25
- Well-known port, 80
- wildcard patterns, *see* patterns
- window activation, 28-30
- window pointer, 28-29, 67
 - with **AUTO** option, 28
 - with **BRKMODE** option, 29
- window titlebar, 67
- Workbench**, 9, 22, 24, 44, 52, 94, 99
 - Amiga-E** commands, 24
 - Application Windows**, 94
 - INFO** tool, 22, 99
- WRAP** **DisplayHandler** option, 28, 32, 41
- writer program, with piping, 69
- WSHAPP**, **AppWindow** tooltype, 95
- WShell-Startup** file, 5, 20-21
- wshell.library**, 3, 6-7, 83
- WshellSeg** file, for **UserShell**, 4, 24
- yank buffer, 37
- "Zoom" gadget, 28



WShell Registration Card

Thank you for purchasing WShell. Please complete the registration form on the opposite side and mail in the detachable card. This will allow us to notify you of software updates or new products. Also, please be sure to inform us of your new address if you move!

We are always interested in improving our software products and try to be responsive to users' needs. If you have suggestions or requests for features that you'd like to see in future releases of the software, please write and tell us.

Prefer to use Electronic Mail? We maintain accounts on some of the major commercial networks, including BIX (WHAWES) and Compuserve (72230,267). From the InterNet you can send mail messages to 72230.267@compuserve.com through the Compuserve gateway. We welcome all inquiries and will try to respond promptly.

Tell A Friend About Us!

If you enjoy using our software products, please consider recommending them to your friends with similar interests. We rely heavily on word-of-mouth advertising in order to keep our costs low and prices reasonable. Referrals from friends usually carry much greater weight than other forms of advertising, and a growing customer base helps ensure the resources for continuing product improvements and support.



WShell 2.0

WShell 2.0 is the most advanced command shell available for the Amiga[®] computer. It provides the shell features you've always wanted, including console-window menus, a session history with a scrollbar, filename completion, and command piping. And yet WShell is highly compatible with the Amiga's system shell, so you won't need to learn a new command language to use it. Among the many features offered by WShell are:

- Superior Line-Editing and Command History
- Console Session History with Scrollbar
- User-Configurable Console-Window Menus
- Command Aliases and Abbreviations
- Resident and Built-In Commands
- Command Piping Using the Familiar "|" Syntax
- Configurable Window Titlebar and Prompt
- Command and Filename Completion
- Iconic Drag-and-Drop Operations with Workbench
- ARexx Interface to Support REXX-Language Macros
- ExecIO Utility for Advanced ARexx Operations

The WShell package also includes the PathHandler, a novel utility that allows you distribute directories like LIBS: and FONTS: across multiple volumes and helps simplify many installation procedures.

WShell 2.0 has been extensively tested to ensure reliable operation, and is superbly documented in a 100+ page indexed manual. WShell will quickly become one of your most essential software applications!

Ask your dealer for our companion product ARexx, the REXX language for the Amiga. ARexx and WShell work together to give you a full procedural language for writing command scripts, offering you unprecedented power and flexibility in a command shell. ARexx is now used as the macro extension language by dozens of other Amiga software products, and has been licensed by Commodore for use with AmigaDOS 2.0!

Developed and Supported by:
Wishful Thinking Development Corp.
P.O. Box 308
Maynard, MA 01754
(508) 568-8695

System requirements: Amiga 500/1000/2000/3000 with V1.3 or V2.0 OS
ARexx and WShell are trademarks of Wishful Thinking Development Corp.
Amiga is a registered trademark of Commodore-Amiga, Inc.